Cours de bases de données – Modèles et langages

Version Septembre 2023

Philippe Rigaux

oct. 03, 2023

Table des matières

1 Introduction **3**
- 1.1 Contenu et plan du cours . . . 3
- 1.2 Apprendre avec ce cours . . . 4
- 1.3 S1 : notions de base . . . 4
  - 1.3.1 Données, bases de données et SGBD . . . 5
  - 1.3.2 Modèle et couches d'abstraction . . . 8
  - 1.3.3 Les langages . . . 10
  - 1.3.4 Quiz . . . 11
- 1.4 Atelier : installation d'un SGBD . . . 11
  - 1.4.1 MySQL . . . 11
  - 1.4.2 Autres . . . 12

2 Le modèle relationnel **13**
- 2.1 S1 : relations et nuplets . . . 14
  - 2.1.1 Qu'est-ce qu'une relation ? . . . 14
  - 2.1.2 Les nuplets . . . 15
  - 2.1.3 Le schéma . . . 15
  - 2.1.4 Mais que représente une relation ? . . . 17
  - 2.1.5 Quiz . . . 17
- 2.2 S2 : clés, dépendances et normalisation . . . 17
  - 2.2.1 Qualité d'un schéma relationnel . . . 18
  - 2.2.2 Schémas normalisés . . . 19
  - 2.2.3 La notion de dépendance fonctionnelle . . . 19
  - 2.2.4 Clés . . . 22
  - 2.2.5 Clés étrangères . . . 23
  - 2.2.6 Quiz . . . 25
- 2.3 S3 : deux exemples de schémas normalisés . . . 25
  - 2.3.1 La base des voyageurs . . . 25
  - 2.3.2 La base des films . . . 27
  - 2.3.3 Quiz . . . 29
- 2.4 Exercices . . . 29
- 2.5 Atelier : Une étude de cas . . . 33

2.5.1 Interprétation des dépendances . . . 33
2.5.2 Recherche d'anomalies . . . 34
2.5.3 Calcul des clés . . . 34

3 SQL, langage déclaratif **35**
3.1 S1 : Un peu de logique . . . 36
3.1.1 Le calcul propositionnel . . . 37
3.1.2 Prédicats . . . 38
3.1.3 Collections et quantificateurs . . . 39
3.1.4 Logique et bases de données . . . 41
3.1.5 Quiz . . . 42
3.2 S2 : SQL conjonctif . . . 42
3.2.1 Requête mono-variable . . . 43
3.2.2 Requêtes multi-variables . . . 46
3.2.3 Quiz . . . 50
3.3 S3 : Quantificateurs et négation . . . 50
3.3.1 Le quantificateur `exists` . . . 50
3.3.2 Quantificateurs et négation . . . 52
3.3.3 Quiz . . . 53
3.4 S4 : Conception d'une requête SQL . . . 53
3.4.1 Conception d'une jointure . . . 55
3.4.2 Conception des requêtes imbriquées . . . 58
3.4.3 La disjonction . . . 59
3.4.4 Quiz . . . 60
3.5 Exercices . . . 60
3.6 Atelier : requêtes SQL sur la base Voyageurs . . . 64
3.7 Atelier : requêtes sur la base des films . . . 65
3.8 Atelier : quand faut-il un `distinct` ? . . . 65

4 SQL, langage algébrique **67**
4.1 S1 : Les opérateurs de l'algèbre . . . 68
4.1.1 La projection, π . . . 68
4.1.2 La sélection, σ . . . 69
4.1.3 Le produit cartésien, $\times$ . . . 70
4.1.4 Renommage . . . 72
4.1.5 L'union, $\cup$ . . . 74
4.1.6 La différence, $-$ . . . 75
4.1.7 Quiz . . . 75
4.2 S2 : la jointure . . . 75
4.2.1 L'opérateur $\bowtie$ . . . 76
4.2.2 Résolution des ambiguités . . . 78
4.2.3 Quiz . . . 80
4.3 S3 : Expressions algébriques . . . 80
4.3.1 Sélection généralisée . . . 81
4.3.2 Requêtes conjonctives . . . 81
4.3.3 Requêtes avec $\cup$ et $-$ . . . 83
4.3.4 Complément d'un ensemble . . . 84
4.3.5 Quantification universelle . . . 84

4.3.6 Quiz . . . 85
4.4 S4 : Complément : évaluation et optimisation . . . 85
4.5 Exercices . . . 87

5 SQL, récapitulatif **93**
5.1 S1 : le bloc `select-from-where` . . . 93
5.1.1 La clause `from` . . . 94
5.1.2 La clause `where` . . . 97
5.1.3 Valeurs manquantes : le `null` . . . 98
5.1.4 La clause `select` . . . 100
5.1.5 Jointure interne, jointure externe . . . 102
5.1.6 Tri et élimination de doublons . . . 106
5.1.7 Quiz . . . 107
5.2 S2 : Requêtes et sous-requêtes . . . 107
5.2.1 Requêtes imbriquées . . . 108
5.2.2 Requêtes correlées . . . 110
5.2.3 Requêtes avec négation . . . 112
5.3 S3 : Agrégats . . . 113
5.3.1 La clause `group by` . . . 114
5.3.2 Quelques exemples . . . 115
5.3.3 La clause `having` . . . 117
5.3.4 Quiz . . . 118
5.4 S4 : Mises à jour . . . 118
5.4.1 Insertion . . . 118
5.4.2 Destruction . . . 119
5.4.3 Modification . . . 119

6 Conception d'une base de données **121**
6.1 S1 : La normalisation . . . 121
6.1.1 La décomposition d'un schéma . . . 122
6.1.2 Algorithme de normalisation . . . 124
6.1.3 Une approche globale . . . 124
6.1.4 Quiz . . . 127
6.2 S2 : Le modèle Entité-Association . . . 127
6.2.1 Le schéma de la base *Films* . . . 127
6.2.2 Entités, attributs et identifiants . . . 129
6.2.3 Types d'entités . . . 130
6.2.4 Associations binaires . . . 132
6.2.5 Quiz . . . 136
6.3 S3 : Concepts avancés . . . 136
6.3.1 Entités faibles . . . 136
6.3.2 Associations généralisées . . . 138
6.3.3 Spécialisation . . . 139
6.3.4 Bilan . . . 140
6.3.5 Quiz . . . 141
6.4 S4 : Du schéma E/A au schéma relationnel . . . 141
6.4.1 Application de la normalisation . . . 141
6.4.2 Illustration avec la base des films . . . 143

6.4.3 Associations avec type d'entité faible . . . 145
6.4.4 Spécialisation . . . 146
6.4.5 Quiz . . . 148
6.5 S5 : Un peu de rétro-ingénierie . . . 148
6.5.1 Quelle conception pour la base des immeubles ? . . . 148
6.5.2 Avec entités faibles . . . 148
6.5.3 Avec réification . . . 150
6.6 Exercices . . . 151
6.7 Atelier : étude du cas « Zoo » : le schéma normalisé . . . 154
6.8 Atelier collectif : concevons le système d'information d'un hôpital . . . 154

7 Schémas relationnel 155
7.1 S1 : Création d'un schéma SQL . . . 155
7.1.1 Types SQL . . . 156
7.1.2 Création des tables . . . 157
7.1.3 Contraintes . . . 158
7.1.4 Clés étrangères . . . 160
7.1.5 Quiz . . . 162
7.2 S2 : Compléments . . . 162
7.2.1 La clause `check` . . . 162
7.2.2 Modification du schéma . . . 163
7.2.3 Création d'index . . . 164
7.3 S3 : Les vues . . . 165
7.3.1 Création et interrogation d'une vue . . . 165
7.3.2 Mise à jour d'une vue . . . 167
7.3.3 Quiz . . . 168
7.4 Exercices . . . 168
7.5 Atelier : étude du cas « Zoo », suite et fin . . . 169

8 Procédures et déclencheurs 171
8.1 S1. Procédures stockées . . . 172
8.1.1 Rôle et fonctionnement des procédures stockées . . . 172
8.1.2 Introduction à PL/SQL . . . 174
8.1.3 Syntaxe de PL/SQL . . . 178
8.1.4 Quiz . . . 185
8.2 S2. Les curseurs . . . 185
8.2.1 Déclaration d'un curseur . . . 185
8.2.2 Exécution d'un curseur . . . 186
8.2.3 Les curseurs PL/SQL . . . 188
8.2.4 Quiz . . . 191
8.3 S3. Les déclencheurs . . . 191
8.3.1 Principes des *triggers* . . . 192
8.3.2 Syntaxe . . . 192
8.3.3 Quelques exemples . . . 193
8.3.4 Quiz . . . 194
8.4 Atelier : JDBC (optionnel) . . . 194
8.4.1 Principes de JDBC . . . 194
8.4.2 Premier exemple . . . 195

8.4.3 Exercices . . . 196
8.4.4 Requêtes préparées . . . 196
8.4.5 L'interface `ResultSet` . . . 197

9 Transactions **199**
9.1 S1 : Transactions . . . 200
9.1.1 Notions de base . . . 201
9.1.2 Exécutions concurrentes . . . 203
9.1.3 Propriétés ACID des transactions . . . 205
9.1.4 Quiz . . . 207
9.2 S2 : Pratique des transactions . . . 207
9.2.1 L'application en ligne « Transactions » . . . 208
9.2.2 Quelques expériences avec l'interface en ligne . . . 210
9.2.3 Mise en pratique directe avec un SGBD . . . 212
9.2.4 Quiz . . . 215
9.3 S3 : effets indésirables des transactions concurrentes . . . 215
9.3.1 Défauts de sérialisabilité . . . 215
9.3.2 Défauts de recouvrabilité . . . 220
9.3.3 Quiz . . . 222
9.4 S4 : choisir un niveau d'isolation . . . 222
9.4.1 Les modes d'isolation SQL . . . 223
9.4.2 Le mode `read committed` . . . 224
9.4.3 Le mode `repeatable read` . . . 224
9.4.4 Le mode `serializable` . . . 227
9.4.5 Verrouillage explicite . . . 229
9.4.6 Quiz . . . 232
9.5 Exercices . . . 232
9.6 Atelier : réservons des places pour Philippe . . . 235
9.6.1 Préparation . . . 235
9.6.2 Déroulement . . . 236

10 Une étude de cas **239**
10.1 S1 : Expression des besoins, conception . . . 239
10.1.1 Quiz . . . 242
10.2 S2 : schéma de la base . . . 242
10.2.1 Quiz . . . 244
10.3 S3 : requêtes . . . 244
10.3.1 Quiz . . . 247
10.4 S4 : Programmation (Python) . . . 247
10.4.1 Un programme de lecture . . . 248
10.4.2 Une transaction . . . 249
10.4.3 Quiz . . . 250
10.5 S5 : aspects transactionnels . . . 250
10.5.1 Cas d'une panne . . . 251
10.5.2 Le curseur est-il impacté par une mise à jour ? . . . 251
10.5.3 Transactions simultanées . . . 252
10.5.4 La bonne méthode . . . 253
10.5.5 Quiz . . . 253

10.6 S6 : *mapping* objet-relationnel . . . 253
10.6.1 Quel problème . . . 254
10.6.2 Quelle solution . . . 255
10.6.3 Quiz . . . 258
10.7 Atelier : une application Django . . . 258
10.7.1 Préliminaires . . . 258
10.7.2 L'application `messagerie` . . . 260
10.7.3 L'ORM de Django : le schéma . . . 261
10.7.4 L'ORM de Django : les données . . . 264
10.7.5 Les vues . . . 265
10.8 Atelier : votre étude de cas . . . 268
10.8.1 La modélisation initiale . . . 268
10.8.2 Peut-on faire mieux ? . . . 269
10.8.3 Une instance-échantillon . . . 270
10.8.4 Le schéma relationnel . . . 270
10.8.5 Les `create table` . . . 270
10.8.6 Les vues . . . 271
10.8.7 Interrogation SQL . . . 271
10.8.8 Et pour aller plus loin . . . 271

11 Annales des examens **273**
11.1 Examen blanc, janvier 2019 . . . 273
11.1.1 Conception (5 points) . . . 274
11.1.2 SQL (6 points) . . . 274
11.1.3 Algèbre (5 points) . . . 274
11.1.4 Transactions (4 points) . . . 275
11.1.5 Correction de l'examen . . . 276
11.2 Examen session 2, avril 2019 . . . 279
11.2.1 Conception (6 points) . . . 279
11.2.2 SQL (8 points) . . . 279
11.2.3 Algèbre (3 points) . . . 280
11.2.4 Programmation et transactions (3 points) . . . 280
11.2.5 Corrigé . . . 281
11.3 Examen session 1, présentiel, juillet 2019 . . . 283
11.3.1 Conception (6 points) . . . 283
11.3.2 SQL (8 points) . . . 283
11.3.3 Algèbre (3 points) . . . 284
11.3.4 Valeurs nulles (2 pts) . . . 284
11.3.5 Transactions (1 pt) . . . 284
11.4 Examen session 2, présentiel, septembre 2020 . . . 284
11.4.1 Schéma relationnel (6 points) . . . 285
11.4.2 SQL (7 points) . . . 285
11.4.3 Un peu de logique (3 points) . . . 285
11.4.4 Algèbre (2 points) . . . 285
11.4.5 Transactions (2 points) . . . 286
11.5 Examen session 1, FOAD, janvier 2022 . . . 286
11.5.1 Conception (8 points) . . . 286
11.5.2 SQL (7 points) . . . 287

11.5.3 Algèbre relationnelle (3 pts) . . . 287
11.5.4 Jointure externe (2 pts) . . . 288
11.6 Examen session 1, second semestre 2023 . . . 288
11.6.1 Conception du schéma (7 pts) . . . 288
11.6.2 SQL (7 pts) . . . 289
11.6.3 Algèbre (2 pts) . . . 290
11.6.4 Valeurs nulles, vues (2 pts) . . . 290
11.6.5 Procédures (2 pts) . . . 290

12 Indices and tables **291**

Contents : Le document que vous commencez à lire fait partie de l'ensemble des supports d'apprentissage proposés sur le site http://www.bdpedia.fr. Il constitue, sous le titre de « Modèles et langages », la première partie d'un cours complet consacré aux bases de données relationnelles.

— La version en ligne du présent support est accessible à http://sql.bdpedia.fr, la version imprimable (PDF) est disponible à http://sql.bdpedia.fr/files/cbd-sql.pdf, et la version pour liseuse / tablette est disponible à http://sql.bdpedia.fr/files/cbd-sql.epub (format EPUB).

— La seconde partie, intitulée « Aspects système », est accessible à http://sys.bdpedia.fr (HTML), http://sys.bdpedia.fr/files/cbd-sys.pdf (PDF) ou http://sys.bdpedia.fr/files/cbd-sys.epub (EPUB).

Je (Philippe Rigaux, Professeur au Cnam) suis également l'auteur de deux autres cours, aux contenus proches :

— Un cours sur les bases de données documentaires et distribuées à http://b3d.bdpedia.fr.

— Un cours sur les applications avec bases de données à http://orm.bdpedia.fr

Reportez-vous à http://www.bdpedia.fr pour plus d'explications.

Important : Ce cours de Philippe Rigaux est mis à disposition selon les termes de la licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International. Cf. http://creativecommons.org/licenses/by-nc-sa/4.0/.

CHAPITRE 1

Introduction

Ce support de cours s'adresse à tous ceux qui veulent *concevoir*, *implanter*, *alimenter* et *interroger* une base de données (BD), et intégrer cette BD à une application. Dans un contexte académique, il s'adresse aux étudiants en troisième année de Licence (L3). Dans un contexte professionnel, le contenu du cours présente tout ce qu'il est nécessaire de maîtriser quand on conçoit une BD ou que l'on développe des applications qui s'appuient sur une BD. Au-delà de ce public principal, toute personne désirant comprendre les principes, méthodes et outils des systèmes de gestion de données trouvera un intérêt à lire les chapitres consacrés à la conception, à l'interrogation et à la programmation SQL, pour ne citer que les principaux.

1.1 Contenu et plan du cours

Le cours est constitué d'un ensemble de chapitres consacrés aux principes et à la mise en œuvre de bases de données *relationnelles*, ainsi qu'à la pratique des Systèmes de Gestion de Bases de Données (SGBD). Il couvre les *modèles et langages* des bases de données, et plus précisément :

- la notion de *modèle de données* qui amène à structurer une base de manière à préserver son intégrité et sa cohérence ;
- la *conception* rigoureuse d'une BD en fonction des besoins d'une application ;
- les *principes* des langages d'interrogation, avec les deux paradigmes principaux : *déclaratif* (on décrit ce que l'on veut obtenir sans dire *comment* on peut l'obtenir) et *procédural* (on applique une suite d'opérations à la base) ; ces deux paradigmes sont étudiés, en pratique, avec le langage SQL ;
- la *mise en pratique* : définition d'un schéma, droits d'accès, insertions et mises à jour ;
- la *programmation* avec une base de données, illustrée avec des langages comme PL/SQL et Python.

Le cours comprend trois parties consacrées successivement au modèle relationnel et à l'interrogation de bases de données relationnelles, à la conception et à l'implantation d'une base, et enfin aux applications s'appuyant sur une base de données avec des éléments de programmation et une introduction aux transactions.

Ce cours ne présente en revanche pas, ou peu, les connaissances nécessaires à la gestion et à l'administration d'une BD : stockage, indexation, évaluation des requêtes, concurrence des accès, reprise sur panne. Ces sujets

sont étudiés en détail dans la seconde partie, disponible séparément sur le site http://sys.bdpedia.fr.

1.2 Apprendre avec ce cours

Le cours est découpé en *chapitres*, couvrant un sujet bien déterminé, et en *sessions*. J'essaie de structurer les sessions pour que les concepts principaux puissent être présentés dans une vidéo d'à peu près 20 minutes. J'estime que chaque session demande environ 2 heures de travail personnel (bien sûr, cela dépend également de vous). Pour assimiler une session vous pouvez combiner les ressources suivantes :

- La lecture du support en ligne : celui que vous avez sous les yeux, également disponible en PDF ou EPUB.
- Le suivi du cours consacré à la session, soit en vidéo, soit en présentiel.
- La réponse au Quiz proposant des QCM sur les principales notions présentées dans la session. Le quiz permet de savoir si vous avez compris : si vous ne savez pas répondre à une question du Quiz, il faut relire le texte, écouter à nouveau la vidéo, approfondir.
- La pratique avec les travaux pratiques en ligne proposés dans plusieurs chapitres.
- Et enfin, la réalisation des exercices proposés en fin de chapitre.

Note : Au Cnam, ce cours est proposé dans un environnement de travail Moodle avec forum, corrections en lignes, interactions avec l'enseignant.

Tout cela constitue autant de manière d'aborder les concepts et techniques présentées. Lisez, écoutez, pratiquez, recommencez autant de fois que nécessaire jusqu'à ce que vous ayez la conviction de maîtriser l'esentiel du sujet abordé. Vous pouvez alors passer à la session suivante.

Les définitions

Pour vous aider à identifier l'essentiel, la partie rédigée du cours contient des définitions. Une définition n'est pas nécessairement difficile, ou compliquée, mais elle est toujours importante. Elle identifie un concept à connaître, et vise à lever toute ambiguité sur l'interprétation de ce concept (c'est comme ça et pas autrement, « par définition »). Apprenez par cœur les définitions, et surtout comprenez-les.

La suite de ce chapitre comprend une unique session avec tout son matériel (vidéos, exercices), consacrée au positionnement du cours.

1.3 S1 : notions de base

Supports complémentaires :

- Diapositives: notions de base
- Vidéo sur les notions de base

Entrons directement dans le vif du sujet avec un premier tour d'horizon qui va nous permettre de situer les principales notions étudiées dans ce cours. Cette session présente sans doute beaucoup de concepts dont

certains s'éclairciront au fur et à mesure de l'avancement dans le cours. À lire et relire régulièrement donc.

1.3.1 Données, bases de données et SGBD

Nous appellerons *donnée* toute valeur numérisée décrivant de manière élémentaire un fait, une mesure, une réalité. Ce peut être une chaîne de caractères (« bouvier »), un entier (365), une date (12/07/1998). Cette valeur est toujours associée au contexte permettant de savoir quelle information elle représente. Un mot comme « bouvier » par exemple peut désigner, entre autres, un gardien de troupeau, un aimable petit insecte, ou le nom d'un écrivain célèbre. Il ne prend un peu de sens que si l'on sait l'interpréter. Une donnée se présente toujours en association avec un contexte interprétatif qui permet de lui donner un sens.

Note : On pourrait établir une distinction (subtile) entre donnée (valeur brute) et information (valeur et contexte interprétatif). Pour ne pas compliquer inutilement les choses, on va assimiler les deux notions dans ce qui suit.

Les données ne tombent pas du ciel, et elles ne sont pas mises en vrac dans un espace de stockage. Elles sont issues d'un domaine applicatif, et décrivent des objets, des faits ou des concepts (on parle plus généralement *d'entités*). On les organise de manière à ce que ces entités soient correctement et uniformément représentées, ainsi que les liens que ces entités ont les unes avec les autres. Si je prends par exemple l'énoncé *Nicolas Bouvier est un écrivain suisse auteur du récit de voyage culte « L'usage du monde » paru en 1963*, je peux en extraire le prénom et le nom d'une personne, sa nationalité (données décrivant une première entité), et au moins un de ses ouvrages (seconde entité, décrite par un titre et une année de parution). J'ai de plus une notion d'auteur qui relie la première à la seconde. Tout cela constitue autant d'informations indissociables les uns des autres, constituant une ébauche d'une base de données consacrée aux écrivains et à leurs œuvres.

La représentation de ces données et leur association donne à la base une *structure* qui aide à distinguer précisément et sans ambiguité les informations élémentaires constituant cette base : nom, prénom, année de naissance, livre(s) publié(s), etc. Une base sans structure n'a aucune utilité. Une base avec une structure incorrecte ou incomplète est une source d'ennuis infinis. Nous verrons comment la structure doit être très sérieusement définie pendant la phase de conception.

Une *base de données* est un ensemble (potentiellement volumineux, mais pas forcément) de telles informations conformes à une structure pré-définie au moment de la conception, avec, de plus, une caractéristique essentielle : on souhaite les mémoriser de manière *persistante*. La persistance désigne la capacité d'une base à exister indépendamment des applications qui la manipulent, ou du système qui l'héberge. On peut arrêter toutes les machines un soir, et retrouver la base de données le lendemain. Cela implique qu'une base est *toujours* stockée sur un support comme les disques magnétiques qui préservent leur contenu même en l'absence d'alimentation électrique.

Important : Les supports persistants (disques, SSD) sont très lents par rapport aux capacités d'un processeur et de sa mémoire interne. La nécessité de stocker une base sur un tel support soulève donc de redoutables problèmes de performance, et a mené à la mise au point de techniques très sophistiquées, caractéristiques des systèmes de gestion de données. Ces techniques sont étudiées dans le cours consacré aux aspects systèmes.

On en arrive donc à la définition suivante :

Définition (base de données)

Une base de données est ensemble d'informations structurées mémorisées sur un support *persistant*.

Remarquons qu'une organisation consistant à stocker nos données dans un (ou plusieurs) fichier(s) sur sur le disque de notre ordinateur personnel peut très bien être considéré comme conforme à cette définition, sous réserve qu'elles soient un tant soit peu structurées. Les fichiers produits par votre traitement de texte préféré par exemple ne font pas l'affaire : on y trouve certes des données, mais pas leur association à un contexte interprétatif non ambigu. Ecrire avec ce traitement de texte une phrase comme « L'usage du monde est un livre de Nicolas Bouvier paru en 1963 » constitue un énoncé trop flou pour qu'un système puisse automatiquement en extraire (sans recourir à des techniques très sophistiquées et en partie incertaines) le nom de l'auteur, le titre de son livre, ou sa date de parution.

Un fichier de base de données a nécessairement une structure qui permet d'une part de distinguer les données les unes des autres, et d'autre part de représenter leurs liens. Prenons l'exemple de l'une des structures les plus simples et les plus répandues, les fichiers CSV. Dans un fichier CSV, les données élémentaires sont réprésentés par des « champs » délimités par des points-virgule. Les champs sont associés les uns aux autres par le simple fait d'être placés dans une même ligne. Les lignes en revanche sont indépendantes les unes des autres. On peut placer autant de lignes que l'on veut dans un fichier, et même changer leur ordre sans que cela modifie en quoi que ce soit l'information représentée.

Voici l'exemple de nos données, représentées en CSV.

```
"Bouvier" ; "Nicolas"; "L'usage du monde" ; 1963
```

On comprend bien que le premier champ est le nom, le second le prénom, etc. Il paraît donc cohérent d'ajouter de nouvelles lignes comme :

```
"Bouvier"   ; "Nicolas"; "L'usage du monde" ; 1963
"Stevenson" ; "Robert-Louis"  ; "Voyage dans les Cévennes avec un âne" ; 1879
```

On a donné une structure régulière à nos informations, ce qui va permettre de les interroger et de les manipuler avec précision. On les stocke dans un fichier sur disque, et nous sommes donc en cours de constitution d'une véritable *base de données*. On peut en fait généraliser ce constat : *une base de données est toujours un ensemble de fichiers, stockés sur une mémoire externe comme un disque, dont le contenu obéit à certaines règles de structuration.*

Peut-on se satisfaire de cette solution et imaginer que nous pouvons construire des applications en nous appuyant directement sur des fichiers structurés, par exemple des fichiers CSV ? C'est la méthode illustrée par la Fig. 1.1. Dans une telle situation, chaque utilisateur applique des programmes au fichier, pour en extraire des données, pour les modifier, pour les créer.

Cette approche n'est pas totalement inenvisageable, mais soulève en pratique de telles difficultés que *personne* (personne de censé en tout cas) n'a recours à une telle solution. Voici un petit catalogue de ces difficultés.

— *Lourdeur d'accès aux données*. En pratique, pour chaque accès, même le plus simple, il faudrait écrire un programme adapté à la structure du fichier. La production et la maintenance de tels programmes seraient extrêmement coûteuses.

— *Risques élevés pour l'intégrité et la sécurité*. Si tout programmeur peut accéder directement aux fichiers, il est impossible de garantir la sécurité et l'intégrité des données. Quelqu'un peut très bien

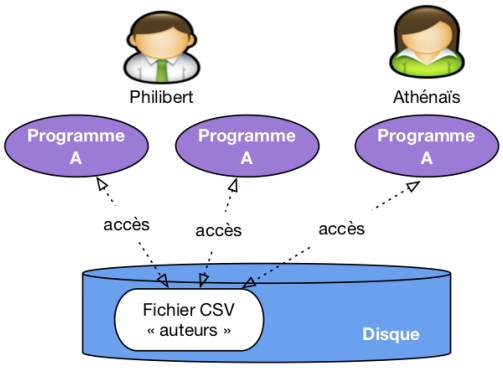

Fig. 1.1 – Une approche simpliste avec accès direct aux fichiers de la base

par exemple, en toute bonne foi, faire une fausse manœuvre qui rend le fichier illisible.

— *Pas de contrôle de concurrence*. Dans un environnement où plusieurs utilisateurs accèdent aux même fichiers, comme illustré par exemple sur la Fig. 1.1, des problèmes de concurrence d'accès se posent, notammment pour les mises à jour. Comment gérer par exemple la situation où deux utilisateurs souhaitent en même temps ajouter une ligne au fichier ?

— *Performances*. Tant qu'un fichier ne contient que quelques centaines de lignes, on peut supposer que les performances ne posent pas de problème, mais que faire quand on atteint les Gigaoctets (1,000 Mégaoctets), ou même le Téraoctet (1,000 Gigaoctets) ? Maintenir des performances acceptables suppose la mise en œuvre d'algorithmes ou de structures de données demandant des compétences très avancées, probablement hors de portée du développeur d'application qui a, de toute façon, mieux à faire.

Chacun de ces problèmes soulève de redoutables difficultés techniques. Leur combinaison nécessite la mise en place de systèmes d'une très grande complexité, capable d'offrir à la fois un accès simple, sécurisé, performant au contenu d'une base, et d'accomplir le tour de force de satisfaire de tels accès pour des dizaines, centaines ou même milliers d'utilisateurs simultanés, le tout en garantissant l'intégrité de la base même en cas de panne. De tels systèmes sont appelés *Systèmes de Gestion de Bases de Données*, SGBD en bref.

Définition (SGBD)

Un Système de Gestion de Bases de Données (SGBD) est un système informatique qui assure la gestion de l'ensemble des informations stockées dans une base de données. Il prend en charge, notamment, les deux grandes fonctionnalités suivantes :

1. Accès aux fichiers de la base, garantissant leur intégrité, contrôlant les opérations concurrentes, optimisant les recherches et mises à jour.
2. Interactions avec les applications et utilisateurs, grâce à des langages d'interrogation et de manipulation à haut niveau d'abstraction.

Avec un SGBD, les applications n'ont plus *jamais* accès directement aux fichiers, et ne savent d'ailleurs même pas qu'ils existent, quelle est leur structure et où ils sont situés. L'architecture classique est celle illustrée par la Fig. 1.2. Le SGBD apparaît sous la forme d'un *serveur*, c'est-à-dire d'un processus informatique prêt à communiquer avec d'autres (les « clients ») via le réseau. Ce serveur est hébergé sur une machine (la « machine serveur ») et est le *seul* à pouvoir accéder aux fichiers contenant les données, ces fichiers étant le plus souvent stockés sur le disque de la machine serveur.

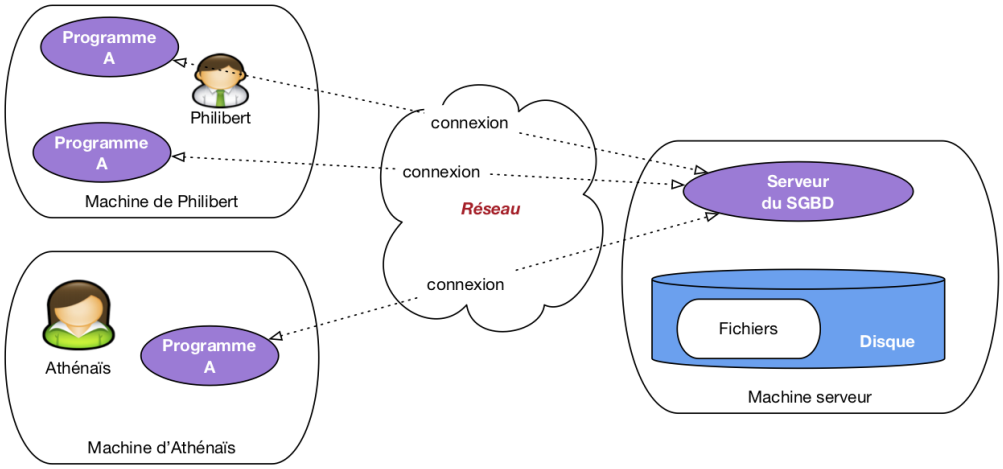

Fig. 1.2 – Architecture classique, avec serveur du SGBD

Les applications utilisateurs, maintenant, accèdent à la base *via* le programme serveur auquel elles sont connectés. Elles transmettent des commandes (d'où le nom « d'applications clientes ») que le serveur se charge d'appliquer. Ces applications bénéficient donc des puissants algorithmes implantés par le SGBD dans son serveur, comme par exemple la capacité à gérer les accès concurrents, où à satisfaire avec efficacité des recherches portant sur de très grosses bases.

Cette architecture est à peu près universellement adoptée par tous les SGBD de tous les temps et de toutes les catégories. Les notions suivantes, et le vocabulaire associé, sont donc très importantes à retenir.

Définition (architecture client serveur)

- **Programme serveur**. Un SGBD est instancié sur une machine sous la forme d'un *programme serveur* qui gère une ou plusieurs bases de données, chacune constituée de fichiers stockés sur disque. Le programme serveur est seul responsable de tous les accès à une base, et de l'utilisation des ressources (mémoire, disques) qui servent de support à ces accès.
- **Clients (programmes)**. Les *programmes (ou applications) clients* se connectent au programme serveur via le réseau, lui transmettent des *requêtes* et recoivent des données en retour. Ils ne disposent d'aucune information directe sur la base.

1.3.2 Modèle et couches d'abstraction

Le fait que le serveur de données s'interpose entre les fichiers et les programmes clients a une conséquence extrêmement importante : ces clients, n'ayant pas accès aux fichiers, ne voient les données que sous la forme que veut bien leur présenter le serveur. Ce dernier peut donc choisir le mode de représentation qui lui semble le plus approprié, la seule condition étant de pouvoir aisément convertir le format des fichiers vers la représentation « publique ».

En d'autres termes, on peut s'abstraire de la complexité et de la lourdeur des formats de fichiers avec tous leurs détails compliqués de codages, de gestion de la mémoire, d'adressage, et proposer une représentation simple et intuitive aux applications. Une des propriétés les plus importantes des SGBD est donc la distinction

entre plusieurs *niveaux d'abstraction* pour la réprésentation des données. Il nous suffira ici de distinguer deux niveaux : le niveau logique et le niveau physique.

Définition : Niveau physique, niveau logique

— Le *niveau physique* est celui du codage des données dans des fichiers stockés sur disque.

— Le *niveau logique* est celui de la représentation les données dans des structures abstraites, proposées aux applications clientes, obtenues par conversion du niveau physique.

Les structures du niveau logique définissent une *modélisation* des données : on peut envisager par exemple des structures de graphe, d'arbre, de listes, etc. Le *modèle relationnel* se caractérise par une modélisation basée sur une seule structure, la table. Cela apporte au modèle une grande simplicité puisque toutes les données ont la même forme et obéissent aux même contraintes. Cela a également quelques inconvénients en limitant la complexité des données représentables. Pour la grande majorité des applications, le modèle relationnel a largement fait la preuve de sa robustesse et de sa capacité d'adaptation. C'est lui que nous étudions dans l'ensemble du cours.

La Fig. 1.3 illustre les niveaux d'abstraction dans l'architecture d'un système de gestion de données. Les programmes clients ne voient que le niveau logique, c'est-à-dire des tables si le modèle de données est relationnel (il en existe d'autres, nous ne les étudions pas ici). Le serveur est en charge du niveau physique, de la conversion des données vers le niveau logique, et de toute la machinerie qui permet de faire fonctionner le système : mémoire, disques, algorithmes et structures de données. Tout cela est, encore une fois, invisible (et c'est tant mieux) pour les programmes clients qui peuvent se concentrer sur l'accès à des données présentées le plus simplement possible.

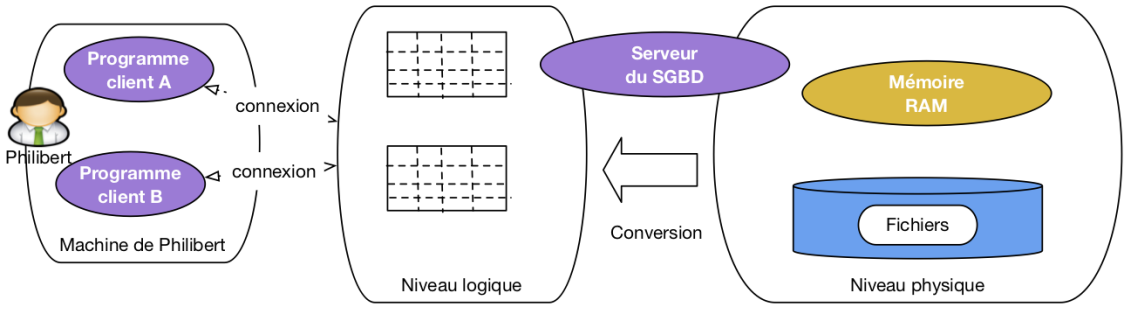

Fig. 1.3 – Illustration des niveaux logique et physique

Signalons pour finir cette courte présentation que les niveaux sont en grande partie indépendants, dans le sens où l'on peut modifier complètement l'organisation du niveau physique sans avoir besoin de changer qui que ce soit aux applications qui accèdent à la base. Cette *indépendance logique-physique* est très précieuse pour l'administration des bases de données.

1.3.3 Les langages

Un modèle, ce n'est pas seulement une ou plusieurs structures pour représenter l'information indépendamment de son format de stockage, c'est aussi un ou plusieurs langages pour interroger et, plus généralement, interagir avec les données (insérer, modifier, détruire, déplacer, protéger, etc.). Le langage permet de construire les commandes transmises au serveur.

Le modèle relationnel s'est construit sur des bases formelles (mathématiques) rigoureuses, ce qui explique en grande partie sa robustesse et sa stabilité depuis l'essentiel des travaux qui l'ont élaboré, dans les années 70-80. Deux langages d'interrogation, à la fois différents, complémentaires et équivalents, ont alors été définis :

1. Un langage *déclaratif*, basé sur la logique mathématique.
2. Un langage *procédural*, et plus précisément *algébrique*, basé sur la théorie des ensembles.

Un langage est *déclaratif* quand il permet de spécifier le résultat que l'on veut obtenir, sans se soucier des opérations nécessaires pour obtenir ce résultat. Un langage algébrique, au contraire, consiste en un ensemble d'opérations permettant de transformer une ou plusieurs tables en entrée en une table - le résultat - en sortie.

Ces deux approches sont très différentes. Elles sont cependant parfaitement complémentaires. l'approche déclarative permet de se concentrer sur le raisonnement, l'expression de requêtes, et fournit une définition rigoureuse de leur signification. L'approche algébrique nous donne une boîte à outil pour calculer les résultats.

Le langage SQL, assemblant les deux approches, a été normalisé sur ces bases. Il est utilisé depuis les années 1970 dans tous les systèmes relationnels, et il paraît tellement naturel et intuitif que même des systèmes construits sur une approche non relationnelle tendent à reprendre ses constructions.

Le terme SQL désigne plus qu'un langage d'interrogation, même s'il s'agit de son principal aspect. La norme couvre également les mises à jour, la définition des tables, les contraintes portant sur les données, les droits d'accès. SQL est donc le langage à connaître pour interagir avec un système relationnel.

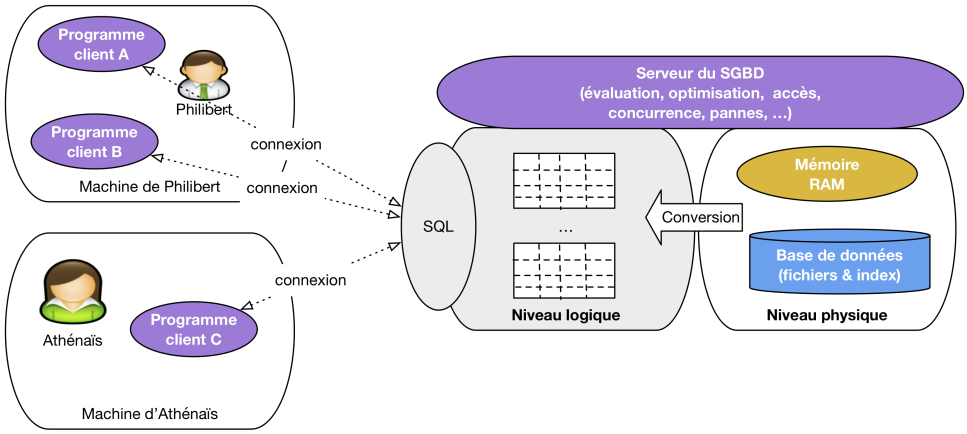

Fig. 1.4 – L'interface « modèle / langage » d'un système relationnel

La Fig. 1.4 étend le schéma précédent en introduisant SQL, qui apparaît comme le constituant central pour établir une communication entre une application et un système relationnel. Les parties grisées de cette figure sont celles couvertes par le cours. Nous allons donc étudier le modèle relationnel (représentation des données sous forme de table), le langage d'interrogation SQL sous ses deux formes, déclarative et algébrique, et l'interaction avec ce langage *via* un langage de programmation permettant de développer des applications.

Tout cela consitue à peu près tout ce qu'il est nécessaire de connaître pour concevoir, implanter, alimenter et interroger une base de données relationnelle, que ce soit directement ou par l'intermédiaire d'un langage de programmtion.

1.3.4 Quiz

1.4 Atelier : installation d'un SGBD

Plutôt que des exercices, ce chapitre peut facilement donner lieu à une mise en pratique basée sur l'installation d'un serveur et d'un client, et de quelques investigations sur leur configuration et leur fonctionnement. Cet atelier n'est bien entendu faisable que si vous disposez d'une ordinateur et d'un minimum de pratique. informatique.

Plusieurs SGBD relationnels sont disponibles en *open source*. Pour chacun, vous pouvez installer le serveur et une ou plusieurs applications clientes.

1.4.1 MySQL

MySQL est un des SGBDR les plus utilisés au monde. Il est maintenant distribué par Oracle Corp, mais on peut l'installer (version MySQL Commiunity server) et l'utiliser gratuitement.

Il est assez pratique d'installer MySQL avec un environnement Web comprenant Apache, le langage PHP et le client phpMyAdmin. Cet environnement est connu sous l'acronyme AMP (Apache - PHP - MySQL).

- Se référer au site http://www.mysql.com pour des informations générales.
- **Choisir une distribution adaptée à votre environnement,**
  - EasyPHP, http://www.easyphp.org/, pour Windows
  - MAMP pour Mac OS X
  - D'innombrables outils pour Linux.
- Suivez les instructions pour installer le serveur et le démarrer.
- Trouvez le fichier de configuration et consultez-le. La configuration d'un serveur comprend typiquement : la mémoire allouée et l'emplacement des fichiers.
- Cherchez où se trouvent les fichiers de base et consultez-les. Peut-on les éditer ou ces fichiers sont-ils en format binaire, lisibles seulement par le serveur ?
- Installez une application cliente. Dans un environnement AMP, vous avez en principe d'office l'application Web phpMyAdmin qui est installée. Essayez de comprendre l'architecture : qui est le serveur, qui est le client, quel est le rôle de Apache, quel est le rôle de votre navigateur web.
- Installez un client autre que phpMyAdmin, par exemple MySQL Workbench (disponible sur le site d'Oracle). Quels sont les paramètres à donner pour connecter ce client au serveur ?
- Executez les scripts SQL de création et d'alimentation de la base de voyageurs.

Quand vous aurez clarifié tout cela pour devriez être en situation confortable pour passer à la suite du cours.

1.4.2 Autres

Si le cœur vous en dit, vous pouvez essayer d'autres systèmes relationnels libres : Postgres, Firebird, Berkeley DB, autres ? À vous de voir.

CHAPITRE 2

Le modèle relationnel

Qu'est-ce donc que ce fameux « modèle relationnel » ? En bref, c'est un ensemble de résultats scientifiques, qui ont en commun de s'appuyer sur une représentation tabulaire des données. Beaucoup de ces résultats ont débouché sur des mises en œuvre pratique. Ils concernent essentiellement deux problématiques complémentaires :

— *La structuration des données*. Comme nous allons le voir dans ce chapitre, on ne peut pas se contenter de placer toute une base de données dans une seule table, sous peine de rencontrer rapidement des problèmes insurmontables. Une base de données relationnelle, c'est un ensemble de tables associées les unes aux autres. La conception du schéma (structures des tables, contraintes sur leur contenu, liens entre tables) doit obéir à certaines règles et satisfaire certaines proprietés. Une théorie solide, la *normalisation* a été développée qui permet de s'assurer que l'on a construit un schéma correct.

— *Les langages d'interrogation.* Le langage SQL que nous connaissons maintenant est issu d'efforts intenses de recherche menés dans les années 70-80. Deux approches se sont dégagées : la principale est une conception *déclarative* des langages de requêtes, basées sur la logique mathématique. Avec cette approche on formule (c'est le mot) ce que l'on souhaite, et le système décide comment calculer le résultat. La seconde est de nature plus procédurale, et identifie l'ensemble minimal des opérateurs dont le système doit disposer pour évaluer une requête. C'est cette seconde approche qui est utilisée en interne pour construire des programmes d'évaluation

Dans ce chapitre nous étudions la structure du modèle relationnel, soit essentiellement la représentation des données, les contraintes, et les règles de normalisation qui définissent la structuration correcte d'une base de données. Deux exemples de bases, commentés, sont donnés en fin de chapitre. Les chapitres suivants seront consacrés aux différents aspects du langage SQL.

2.1 S1 : relations et nuplets

Supports complémentaires :
— Diapositives: modèle relationnel
— Vidéo sur le modèle relationnel

L'expression « modèle relationnel » a pour origine (surprise !) la notion de relation, un des fondements mathématiques sur lesquels s'appuie la théorie relationnelle. Dans le modèle relationnel, la seule structure acceptée pour représenter les données est la relation.

2.1.1 Qu'est-ce qu'une relation ?

Etant donné un ensemble d'objets O, une *relation* (binaire) sur O est un sous-ensemble du produit cartésien $O \times O$. Au cas où vous l'auriez oublié, le produit cartésien entre deux ensembles $A \times B$ est l'ensemble de toutes les paires possibles constituées d'un élément de A et d'un élément de B.

Dans le contexte des bases de données, les objets auxquels on s'intéresse sont des valeurs élémentaires comme les entiers I, les réels (ou plus précisément les nombres en virgule flottante puisqu'on ne sait pas représenter une précision infinie) F, les chaînes de caractères S, les dates, etc. La notion de valeur *élémentaire* s'oppose à celle de valeur *structurée* : il n'est pas possible en relationnel de placer dans une cellule un graphe, une liste, un enregistrement.

On introduit de plus une restriction importante : les relations sont *finies* (on ne peut pas représenter en extension un ensemble infini avec une machine).

L'ensemble des paires constituées des noms de département et et de leur numéro de code est par exemple une relation en base de données : c'est un ensemble fini, sous-ensemble du produit cartésien $S \times I$.

La notion de relation binaire se généralise facilement. Une relation ternaire sur A, B, C est un sous-ensemble fini du produit cartésien $A \times B \times C$, qui lui même s'obtient par $(A \times B) \times C$. On peut ainsi créer des relations de dimension quelconque.

Définition : relation

Une relation de degré *n* sur les domaines $D_1, D_2, \cdots, D_n$ est un sous-ensemble fini du produit cartésien $D_1 \times D_2 \times \cdots \times D_n$

Une relation est un objet abstrait, on peut la représenter de différentes manières. Une représentation naturelle est le graphe comme le montre la Fig. 2.1. Une autre structure possible est la table, qui s'avère beaucoup plus pratique quand la relation n'est plus binaire mais ternaire et au-delà.

nom	code
Ardèche	07
Gard	30
Manche	50
Paris	75

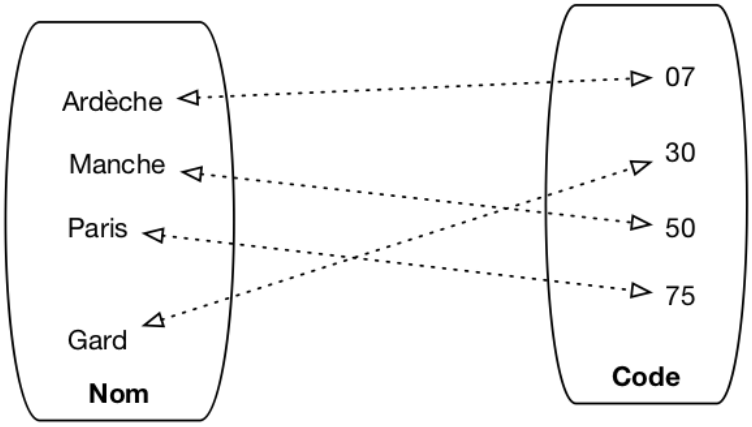

Fig. 2.1 – Une relation binaire représentée comme un graphe

Dans une base relationnelle, on utilise toujours la représentation d'une relation sous forme de table. À partir de maintenant nous pourrons nous permettre d'utiliser les deux termes comme synonymes.

2.1.2 Les nuplets

Un élément d'une relation de dimension *n* est un *nuplet* $(a_1, a_2, \cdots, a_n)$. Dans la représentation par table, un nuplet est une ligne. Là encore nous assimilerons les deux termes, en privilégiant toutefois *nuplet* qui indique plus précisément la structure constituée d'une liste de valeurs.

La définition d'une relation comme un ensemble (au sens mathématique) a quelques conséquences importantes :

— *L'ordre des nuplets est indifférent* car il n'y a pas d'ordre dans un ensemble ; conséquence pratique : le résultat d'une requête appliquée à une relation ne dépend pas de l'ordre des lignes dans la relation.
— *On ne peut pas trouver deux fois le même nuplet* car il n'y a pas de doublons dans un ensemble.
— Il n'y a pas (en théorie) de « cellule vide » dans la relation ; toutes les valeurs de tous les attributs de chaque nuplet sont toujours connues.

Dans la pratique les choses sont un peu différentes pour les doublons et les cellules vides, comme nous le verrons

2.1.3 Le schéma

Et, finalement, on notera qu'aussi bien la représentation par graphe que celle par table incluent un nommage de chaque dimension (le `nom` du département, son `code`, dans notre exemple). Ce nommage n'est pas strictement indispensable (on pourrait utiliser la position par exemple), mais s'avère très pratique et sera donc utilisé systématiquement.

On peut donc *décrire* une relation par

1. Le nom de la relation.
2. Un nom (distinct) pour chaque dimension, dit *nom d'attribut*, noté A_i.
3. Le domaine de valeur (type) de chaque dimension, noté D_i.

Cette description s'écrit de manière concise $R(A_1 : D_1, A_2 : D_2, \cdots, A_n : D_n)$, et on l'appelle le *schéma* de la relation. Tous les A_i sont distincts, mais on peut bien entendu utiliser plusieurs fois le même type. Le schéma de notre table des départements est donc `Département (nom: string, code: string)`. Le domaine de valeur ayant relativement peu d'importance, on pourra souvent l'omettre et écrire le schéma `Département (nom, code)`. Il est d'aileurs relativement facile de changer le type d'un attribut sur une base existante.

Et c'est tout ! Donc en résumé,

Définition : relation, nuplet et schéma

1. Une *relation* de degré *n* sur les domaines $D_1, D_2, \cdots, D_n$ est un sous-ensemble fini du produit cartésien $D_1 \times D_2 \times \cdots \times D_n$.
2. Le schéma d'une relation s'écrit $R(A_1 : D_1, A_2 : D_2, \cdots, A_n : D_n)$, ***R*** étant le nom de la relation et les A_i, deux à deux distincts, les noms d'attributs.
3. Un élément de cette relation est un *nuplet* $(a_1, a_2, \cdots, a_n)$, les a_i étant les valeurs des attributs.

Et en ce qui concerne le vocabulaire, le tableau suivant montre celui, rigoureux, issu de la modélisation mathématique et celui, plus vague, correspondant à la représentation par table. Les termes de chaque ligne seront considérés comme équivalents, mais on privilégiera les premiers qui sont plus précis.

Terme du modèle	Terme de la représentation par table
Relation	Table
nuplet	ligne
Nom d'attribut	Nom de colonne
Valeur d'attribut	Cellule
Domaine	Type

Attention à utiliser ce vocabulaire soigneusement, sous peine de confusion. Ne pas confondre par exemple le nom d'attribut (qui est commun à toute la table) et la valeur d'attribut (qui est spécifique à un nuplet).

La structure utilisée pour représenter les données est donc extrêmement simple. Il faut insister sur le fait que les valeurs des attributs, celles que l'on trouve dans chaque cellule de la table, sont élémentaires : entiers, chaînes de caractères, etc. On *ne peut pas* avoir une valeur d'attribut qui soit un tant soit peu construite, comme par exemple une liste, ou une sous-relation. Les valeurs dans une base de données sont dites *atomiques* (pour signifier qu'elles sont non-décomposables, rien de toxique à priori). Cette contrainte conditionne tous les autres aspects du modèle relationnel, et notamment la conception, et l'interrogation.

Une base bien formée suit des règles dites de normalisation. La forme normale minimale est définie ci-dessous.

Définition : première forme normale

Une relation est en première forme normale si toutes les valeurs d'attribut sont connues et atomiques et si elle ne contient aucun doublon.

Un doublon n'apporte aucune information supplémentaire et on les évite donc. En pratique, on le fait en ajoutant des critères d'unicité sur certains attributs, la *clé*.

On considère pour l'instant que *toutes* les valeurs d'un nuplet sont connues. En pratique, c'est une contrainte trop forte que l'on sera amené à lever avec SQL, au prix de quelques difficultés supplémentaires.

2.1.4 Mais que représente une relation ?

En première approche, une relation est simplement un ensemble de nuplets. On peut donc lui appliquer des opérations ensemblistes : intersection, union, produit cartésien, projection, etc. Cette vision se soucie peu de la signification de ce qui est représenté, et peut mener à des manipulations dont la finalité reste obscure. Ce n'est pas forcément le meilleur choix pour un utilisateur humain, mais ça l'est pour un système qui ne se soucie que de la description opérationnelle.

Dans une seconde approche, plus « sémantique », une relation est un mécanisme permettant d'énoncer des faits sur le monde réel. Chaque nuplet correspond à un tel énoncé. Si un nuplet est présent dans la relation, le fait est considéré comme vrai, sinon il est faux.

La table des départements sera ainsi interprétée comme un ensemble d'énoncés : « Le département de l'Ardèche a pour code 07 », « Le département du Gard a pour code 30 », et ainsi de suite. Si un nuplet, par exemple, `(Gers 32)`, n'est pas dans la base, on considère que l'énoncé « Le département du Gers a pour code 32 » est faux.

Cette approche mène directement à une manipulation des données fondée sur des raisonnements s'appuyant sur les valeurs de vérité énoncées par les faits de la base. On a alors recours à la logique formelle pour exprimer ces raisonnements de manière rigoureuse. Dans cette approche, qui est à la base de SQL, interroger une base, c'est déduire un ensemble de faits qui satisfont un énoncé logique (une « formule »). Selon ce point de vue, SQL est un langage pour écrire des formules logiques, et un système relationnel est (entre autres) une machine qui effectue des démonstrations.

2.1.5 Quiz

2.2 S2 : clés, dépendances et normalisation

Supports complémentaires :

— Diapositives: clés/dépendances
— Vidéo sur les clés/dépendances

Comme nous l'avons vu ci-dessus, le schéma d'une relation consiste – pour l'essentiel – en un nom (de relation) et un ensemble de noms d'attributs. On pourrait naïvement penser qu'il suffit de créer une unique relation et de tout mettre dedans pour avoir une base de données. En fait, une telle approche est inapplicable et il est indispensable de créer plusieurs relations, associées les unes aux autres.

Le *schéma d'une base de données* est donc constitué d'un ensemble de schéma de relations. Pourquoi en arrive-t-on là et quels sont les problèmes que l'on souhaite éviter ? C'est ce que nous étudions dans cette session. La notion centrale introduite ici est celle de *clé* d'une relation.

2.2.1 Qualité d'un schéma relationnel

Voici un exemple de schéma, avec une notation très simplifiée, que nous allons utiliser pour discuter de la notion centrale de « bon » et « mauvais » schéma. On veut créer une base de données représentant des films, avec des informations comme le titre, l'année, le metteur en scène, etc. On part d'un schéma rassemblant ces informations dans une unique table :

```
Film(titre, année, prénomRéalisateur, nomRéalisateur, annéeNaiss)
```

Un tel schéma permet-il de gérer correctement les données ? Regardons un exemple de contenu de la table.

titre	année	prénomRéalisateur	nomRéalisateur	année-Nais
Alien	1979	Ridley	Scott	1943
Vertigo	1958	Alfred	Hitchcock	1899
Psychose	1960	Alfred	Hitchcock	1899
Kagemusha	1980	Akira	Kurosawa	1910
Volte-face	1997	John	Woo	1946
Pulp Fiction	1995	Quentin	Tarantino	1963
Titanic	1997	James	Cameron	1954
Sacrifice	1986	Andrei	Tarkovski	1932

Même pour une information aussi simple, il est facile d'énumérer tout un ensemble de problèmes potentiels. Tous ou presque découlent d'un grave défaut de la table ci-dessus : *il est possible de représenter la même information plusieurs fois*, ou, pour employer un mot que nous retrouverons souvent, *il y a redondance de l'information*.

Anomalies lors d'une insertion

Rien n'empêche de représenter plusieurs fois le même film. Pire : il est possible d'insérer plusieurs fois le film *Vertigo* en le décrivant à chaque fois de manière différente, par exemple en lui attribuant une fois comme réalisateur Alfred Hitchcock, puis une autre fois John Woo, etc.

La bonne question consiste d'ailleurs à se demander ce qui distingue deux films l'un de l'autre, et à quel moment on peut dire que la même information a été répétée. Peut-il y avoir deux films différents avec le même titre par exemple ? Si la réponse est non (?), alors on devrait pouvoir assurer qu'il n'y a pas deux lignes dans la table avec la même valeur pour l'attribut *titre*. Si la réponse est oui (ce qui semble raisonnable), il reste à déterminer quel est l'ensemble des attributs qui permet de caractériser de manière unique un film ou, à défaut, de créer un tel *identifiant* artificiellement. C'est une notion centrale et délicate sur laquelle nous revenons de manière approfondie ultérieurement.

Autre anomalie liées aux insertions : on ne peut pas insérer un film si on ne connaît pas son metteur en scène et réciproquement.

Anomalies lors d'une modification

La redondance d'information entraîne également des anomalies de mise à jour. Supposons que l'on modifie l'année de naissance de Hitchcock pour la ligne *Vertigo* et pas pour la ligne *Psychose*. On se retrouve alors avec des informations incohérentes. Les mêmes questions que précédemment se posent d'ailleurs. Jusqu'à

quel point peut-on dire qu'il n'y a qu'un seul réalisateur nommé Hitchcock, et qu'il ne doit donc y avoir qu'une seule année de naissance pour un réalisateur de ce nom ?

Anomalies lors d'une destruction

On ne peut pas supprimer un film sans supprimer du même coup son metteur en scène. Si on souhaite, par exemple, ne plus voir le film *Titanic* figurer dans la base de données, on va effacer du même coup les informations sur James Cameron.

2.2.2 Schémas normalisés

Que déduire de ce qui précède ? Tout d'abord qu'il existe des schémas avec de bonnes propriétés, et d'autres qui souffrent de défauts de conception, lesquels entraînent de sérieux problèmes de gestion de la base. Ensuite, que nous avons besoin d'aller plus loin qu'une simple énumération d'attributs et énoncer des *contraintes* et des *règles* qui nous indiquent plus précisément les liens qui caractérisent les données.

Le modèle relationnel nous propose un outil précieux pour répondre à ces questions : la *normalisation*. Un schéma normalisé présente des caractéristiques formelles qu'il est possible d'évaluer. La normalisation nous garantit l'absence de défaut (et notamment de redondance) tout en préservant l'intégralité de l'information représentée.

La théorie du modèle relationnel a développé une construction formelle solide pour qualifier les propriétés d'un schéma d'une part, et décomposer un schéma dénormalisé en schéma normalisé d'autre part. Le premier, détaillé ci-dessous, donne un éclairage très précis sur ce qu'est un bon schéma relationnel. Le second aspect fait l'objet du chapitre *Conception d'une base de données*.

2.2.3 La notion de dépendance fonctionnelle

Le principal concept est celui de dépendance fonctionnelle, qui fournit une construction de base pour élaborer les contraintes dont nous avons besoin pour caractériser nos données et leurs liens. Il s'énonce comme suit.

Définition : dépendance fonctionnelle

Soit un schéma de relation *R*, *S* un *sous-ensemble* d'attributs de *R*, et *A* un attribut quelconque de *R*.

On dit que *A* *dépend fonctionnellement* de *S* (ce que l'on note $S \rightarrow A$) quand, pour toute paire (l_1, l_2) de lignes de *R*, l'égalité de l_1 et de l_2 sur *S* implique l'égalité sur *A*.

Informellement, on peut raisonner ainsi : « la valeur de *S* détermine la valeur de *A* », ou encore « Si je connais *S*, alors je connais *A* ». Tout se passe comme s'il existait une fonction qui, étant donnée une valeur de *S*, produit la valeur de *A* (toujours la même, par définition d'une fonction). Par, exemple, si je prends la relation `Personne` avec l'ensemble des attributs suivants

```
(nom, prénom, noSS, dateNaissance, adresse, email)
```

je peux considérer les dépendances fonctionnelles suivantes :

— $email \rightarrow nom, prnom, noSS, dateNaissance, adresse$
— $noSS \rightarrow email, nom, prnom, dateNaissance, adresse$

J'ai donc considéré que la connaisance d'une adresse électronique détermine la connaissance des valeurs des autres attributs, et de même pour le numéro de sécurité sociale.

Note : La notation $S \to A, B$ est un racourci pour $S \to A$ et $S \to B$

On peut avoir des dépendances fonctionnelles où la partie gauche comprend plusieurs attributs. Par exemple, pour les attributs suivants :

```
noEtudiant, noCours, année, note, titreCours
```

on peut énoncer la dépendance fonctionnelle suivante :

$$noEtudiant, noCours, anne \to note, titreCours$$

La connaissance d'un étudiant, d'un cours et d'une année détermine la note obtenue et le titre du cours.

Prenons quelques exemples. Le tableau suivant montre une relation `R(A1, A2, A3, A4)`.

A1	A2	A3	A4
1	2	3	4
1	2	3	5
6	7	8	2
2	1	3	4

Les dépendances fonctionnelles suivantes sont respectées :
- $A_1 \to A_3$
- $A_2, A_3 \to A_1$
- $A_4 \to A_3$

En revanche les suivantes sont violées : $A_4 \to A_1$, $A_2, A_3 \to A_4$.

Certaines propriétés fondamentales des DFs (les axiomes d'Armstrong) sont importantes à connaître.

Axiomes d'Armstrong
- Réflexivité : si $A \subseteq X$, alors $X \to A$. C'est une propriété assez triviale : si je connais X, alors je connais toute partie de X.
- Augmentation : si $X \to Y$, alors $XZ \to Y$ pour tout Z. Là aussi, c'est assez trivial : si la connaissance de X détermine Y, alors la connaissance d'un sur-ensemble de X détermine *à plus forte raison* Y.
- Transitivité : si $X \to Y$ et si $Y \to Z$, alors $X \to Z$. Si X détermine Y et Y détermine Z, alors X détermine Z.

Reprenons l'exemple suivant :

$$noEtudiant, noCours, anne \to note, titreCours$$

Nous avons ici l'illustration d'une dépendance fonctionnelle obtenue par transitivité. En effet, on peut admettre la dépendance suivante :

$$noCours \to titreCours$$

Dans ce cas, connaissant les 3 valeurs du nuplet `(noEtudiant, noCours, année)`, je connais la valeur de `noCours` (réflexivité) , et connaissant le numéro du cours je connais le titre du cours. La connaissance du titre à partir de la clé est obtenue par transitivité.

On se restreint pour l'étude de la normalisation aux DF *minimales* et *directes*.

Définition : dépendances minimales et directes

Une dépendance fonctionnelle $A \rightarrow X$ est *minimale* s'il n'existe pas d'ensemble d'attributs $B \subset A$ tel que $B \rightarrow X$.

Une dépendance fonctionnelle $A \rightarrow X$ est *directe* si elle n'est pas obtenue par transitivité.

Les dépendances fonctionnelles fournissent un outil pour analyser la qualité d'un schéma relationnel. Prenons le cas d'un système permettant d'évaluer des manuscrits soumis à un éditeur. Voici deux schémas possibles pour représenter les rapports produits par des experts.

- **Schéma 1**
  - Manuscrit (id_manuscrit, auteur, titre, id_expert, nom, commentaire)
- **Schéma 2**
  - Manuscrit (id_manuscrit, auteur, titre, id_expert, commentaire)
  - Expert (id_expert, nom)

Et on donne les dépendances fonctionnelles minimales et directes suivantes :

- $id_manuscrit \rightarrow auteur, titre, id_expert, commentaire$
- $id_expert \rightarrow nom$

On suppose donc qu'il existe un seul expert par manuscrit. Ces dépendances nous donnent un moyen de caractériser précisément les redondances et incohérences potentielles. Voici un exemple de relation pour le schéma 1.

id_manuscrit	auteur	titre	id_expert	nom	commentaire
10	Serge	L'arpète	2	Philippe	Une réussite, on tourne les pages avec frénésie
20	Cécile	Un art du chant grégorien sous Louis XIV	2	Sophie	Un livre qui fait date sur le sujet. Bravo
10	Serge	L'arpète	2	Philippe	Une réussite, on tourne les pages avec frénésie
10	Philippe	SQL	1	Sophie	la référence

En nous basant sur les dépendances fonctionnelles associées à ce schéma on peut énumérer les anomalies suivantes :

- La DF $id_expert \rightarrow nom$ n'est pas respectée par le premier et deuxième nuplet. Pour le même `id_expert`, on trouve une fois le nom « Philippe », une fois le nom « Sophie ».
  En revanche cette DF est respectée si on ne considère que le premier, le troisième et le quatrième nuplet.
- La DF $id_manuscrit \rightarrow auteur, titre, id_expert, commentaire$ n'est pas respectée par le premier et quatrième nuplet. Pour le même `id_manuscrit`, on trouve des valeurs complètement différentes.

En revanche cette DF est respectée par le premier et troisième nuplet, et on constate une totale redondance : ces nuplets sont des doublons.

En résumé, on a soit des redondances, soit des incohérences. *Il est impératif d'éviter toutes ces anomalies*.

On pourrait envisager de demander à un SGBD de considérer les DFs comme des *contraintes* sur le contenu de la base de données et d'assurer leur préservation. On éliminerait les incohérences mais pas les redondances. De plus le contrôle de ces contraintes serait, d'évidence, très coûteux. Il existe une bien meilleure solution, basée sur les clés et la décomposition des schémas.

2.2.4 Clés

Commençons par définir la notion essentielle de *clé*.

Définition : clé

Une clé d'une relation R est un sous-ensemble *minimal* C des attributs tel que tout attribut de R dépend fonctionnellement de C.

L'attribut `id_expert` est une clé de la relation *Expert* dans le schéma 2. Dans le schéma 1, l'attribut `id_manuscrit` est une clé de *Manuscrit*. Notez que tout attribut de la relation dépend *aussi* de la paire `(id_manuscrit, auteur)`, sans que cette paire soit une clé puisqu'elle n'est pas *minimale* (il existe un sous-ensemble strict qui est lui-même clé).

Note : Comme le montre l'exemple de la relation `Personne` ci-dessus, on peut en principe trouver plusieurs clés dans une relation. On en choisit alors une comme *clé primaire*.

Et maintenant nous pouvons définir ce qu'est un schéma de relation normalisé.

Définition : schéma normalisé (troisième forme normale)

Un schéma de relation R est *normalisé* quand, dans toute dépendance fonctionnelle (minimale et directe) $S \to A$ sur les attributs de R, S est une clé.

Remarque

Cette définition est celle de la forme normale dite « de Boyce-Codd ». La définition standard de la troisième forme normale est un peu moins stricte (et un peu plus difficile à saisir intuitivement) : elle demande que tout attribut non-clé soit dépendant fonctionnellement d'une clé.

La différence est subtile et très rarement rencontrée en pratique : la troisième forme normale autorise une DF d'un attribut non-clé vers *une partie* de la clé, alors que la version de Boyce-Codd exclut ce cas.

En toute rigueur, il faudrait connaître et discuter des deux versions de la définition mais, le gain pratique étant négligeable, j'assume de vous demander de comprendre et de retenir la définition la plus simple et la plus intuitive.

La relation `Manuscrit` dans le schéma 1 ci-dessus n'est pas normalisée à cause de la dépendance fonctionnelle `id_expert` $\rightarrow$ `nom`, alors que l'attribut `id_expert` n'est pas une clé. Il existe une version intuitive de cette constatation abstraite : la relation `Manuscrit` contient des informations qui ne sont pas *directement* liées à la notion de manuscrit. La présence d'informations indirectes est une source de redondance et donc d'anomalies.

L'essentiel de ce qu'il faut comprendre est énoncé dans ce qui précède. On veut obtenir des relations normalisées car il et facile de montrer que la dénormalisation entraîne toutes sortes d'anomalies au moment où la base est mise à jour. De plus, si *R* est une relation de clé *C*, deux lignes de *R* ayant les même valeurs pour *C* auront par définition les mêmes valeurs pour les autres attributs et seront donc parfaitement identiques. Il est donc inutile (et nuisible) d'autoriser cette situation : on fera en sorte que la valeur d'une clé soit *unique* pour l'ensemble des lignes d'une relation. En résumé on veut des schémas de relation normalisés et dotés d'une clé unique bien identifiée. Cette combinaison interdit toute redondance.

Note : Plusieurs formes de normalisation ont été proposées. Celle présentée ici est dite « troisième forme normale » (3FN). Il est toujours possible de se ramener à des relations en 3FN.

2.2.5 Clés étrangères

Un bon schéma relationnel est donc un schéma où toutes les tables sont normalisées. Cela signifie que, par rapport à notre approche initiale naïve où toutes les données étaient placées dans une seule table, nous devons *décomposer* cette unique table en fonction des clés.

Prenons notre second schéma.

— Manuscrit (id_manuscrit, auteur, titre, id_expert, commentaire)
— Expert (id_expert, nom)

Ces deux relations sont normalisées, avec pour clés respectives `id_manuscrit` et `id_expert`. On constate que `id_expert` est présent dans les *deux* schémas. Ce n'est pas une clé de la relation `Manuscrit`, mais c'est la duplication de la clé de `Expert` dans `Manuscrit`. Quelle est son rôle ? Le raisonnement est exactement le suivant :

— `id_expert` est la clé de `Expert` : connaissant `id_expert`, je connais donc aussi (par définition) toutes les autres informations sur l'expert.
— `id_manuscrit` est la clé de `Manuscrit` : connaissant `id_manuscrit`, je connais donc aussi (par définition) toutes les autres informations sur le manuscrit, et notamment `id_expert`.
— Et donc, par transitivité, connaissant `id_manuscrit`, je connais `id_expert`, et connaissant `id_expert`, je connais toutes les autres informations sur l'expert : je n'ai perdu aucune information en effectuant la décomposition puisque les dépendances me permettent de reconstituer la situation initiale.

L'attribut `id_expert` dans la relation `Manuscrit` est une *clé étrangère*. Une clé étrangère permet, par transitivité, de tout savoir sur le nuplet identifié par sa valeur, ce nuplet étant en général (pas toujours) placé dans une autre table.

Définition : clé étrangère

Soit R et S deux relations de clés (primaires) respectives `idR` et `idS`. Une *clé étrangère* de S dans R est un attribut `ce` de R dont la valeur est *toujours* identique à (exactement) une des valeurs de `idS`.

Intuitivement, ce « référence » un (et un seul) nuplet de S.

Voici une illustration du mécanisme de clé primaire et de clé étrangère, toujours sur notre exemple de manuscrit et d'expert. Prenons tout d'abord la table des experts.

id_expert	nom	adresse
1	Sophie	rue Montorgueil, Paris
2	Philippe	rue des Martyrs, Paris

Et voici la table des manuscrits. Rappelons que `id_expert` est la clé étrangère de `Expert` dans `Manuscrit`.

id_manuscrit	auteur	titre	id_expert	commentaire
10	Serge	L'arpète	2	Une réussite, on tourne les pages avec frénésie
20	Cécile	Un art du chant grégorien sous Louis XIV	1	Un livre qui fait date sur le sujet. Bravo

Voyez-vous quel(le) expert(e) a évalué quel manuscrit ? Etes-vous d'accord que connaissant la valeur de clé d'un manuscrit, je connais sans ambiguité le nom de l'expert qui l'a évalué ? Constatez-vous que ces relations sont bien normalisées ?

Une clé étrangère ne peut prendre ses valeurs que dans l'ensemble des valeurs de la clé référencée. Dans notre exemple, la valeur de la clé étrangère `id_expert` dans `Manuscrit` est impérativement l'une des valeurs de clé de `id_expert`. Si ce n'était pas le cas, on ferait référence à un expert qui n'existe pas.

Dans un schéma normalisé, un système doit donc gérer deux types de contraintes, toutes deux liées aux clés.

Définition : contraintes d'unicité, contrainte d'intégrité référentielle.

Contrainte d'unicité : une valeur de clé ne peut apparaître qu'une fois dans une relation.

Contrainte d'intégrité référentielle : la valeur d'une clé étrangère doit *toujours* être également une des valeurs de la clé référencée.

Ces deux contraintes garantissent l'absence totale de redondances et d'incohérences. La session suivante va commenter deux exemples complets. Quant à la démarche complète de conception, elle sera développée dans le chapitre *Conception d'une base de données*.

2.2.6 Quiz

2.3 S3 : deux exemples de schémas normalisés

Supports complémentaires :

- Diapositives: deux schémas normalisés
- Vidéo sur les schémas normalisés
- Schéma de la base des voyageurs et base des voyageurs (si vous souhaitez les installer dans votre environnement).
- Schéma de la base des films et base des films (si vous souhaitez les installer dans votre environnement).

Dans l'ensemble du cours nous allons utiliser quelques bases de données, petites, simples, à des fins d'illustration, pour les langages d'interrogation notamment. Elles sont présentées ci-dessous, avec quelques commentaires sur le schéma, que nous considérons comme donné pour l'instant. Si vous vous demandez par quelle méthode on en est arrivé à ces schémas, reportez-vous au chapitre *Conception d'une base de données*.

2.3.1 La base des voyageurs

Notre première base de données décrit les pérégrinations de quelques voyageurs plus ou moins célèbres. Ces voyageurs occupent occasionnellement des logements pendant des périodes plus ou moins longues, et y exercent (ou pas) quelques activités.

Voici le schéma de la base. Les clés primaires sont en **gras**, les clés étrangères en *italiques*. Essayez de vous figurer les dépendances fonctionnelles et la manière dont elles permettent de rassembler des informations réparties dans plusieurs tables.

- Voyageur (**idVoyageur**, nom, prénom, ville, région)
- Séjour (**idSéjour**, *idVoyageur*, *codeLogement*, début, fin)
- Logement (**code**, nom, capacité, type, lieu)
- Activité (**codeLogement, codeActivité**, description)

La table des voyageurs

La table `Voyageur` ne comprend aucune clé étrangère. Les voyageurs sont identifiés par un numéro séquentiel nommé `idVoyageur`, incrémenté de 10 en 10 (on aurait pu incrémenter de 5, ou de 100, ou changer à chaque fois : la seule chose qui compte est que chaque identifiant soit unique). On indique la ville et la région de résidence.

id-Voyageur	nom	prénom	ville	région
10	Fogg	Phileas	Ajaccio	Corse
20	Bouvier	Nicolas	Aurillac	Auvergne
30	David-Néel	Alexandra	Lhassa	Tibet
40	Stevenson	Robert Louis	Vannes	Bretagne

Remarquez que nos régions ne sont pas des régions administratives au sens strict : cette base va nous permettre d'illustrer l'interrogation de bases relationnelles, elle n'a aucune prétention à l'exatitude.

La table Logement

La table `Logement` est également très simple, son schéma ne contient pas de clé étrangère. La clé est un code synthétisant le nom du logement. Voici son contenu.

code	nom	capa-cité	type	lieu
pi	U Pinzutu	10	Gîte	Corse
ta	Tabriz	34	Hôtel	Bretagne
ca	Causses	45	Auberge	Cévennes
ge	Génépi	134	Hôtel	Alpes

L'information nommée `région` dans la table des voyageurs d'appelle maintenant `lieu` dans la table `Logement`. Ce n'est pas tout à fait cohérent, mais corrrespond à des situations couramment rencontrées où la même information apparaît sous des noms différents. Nous verrons que le modèle relationnel est équipé pour y faire face.

La table des séjours

Les séjours sont identifiés par un numéro séquentiel incrémenté par unités. Le début et la fin sont des numéros de semaine dans l'année (on fait simple, ce n'est pas une base pour de vrai).

idSéjour	idVoyageur	codeLogement	début	fin
1	10	pi	20	20
2	20	ta	21	22
3	30	ge	2	3
4	20	pi	19	23
5	20	ge	22	24
6	10	pi	10	12
7	30	ca	13	18
8	20	ca	21	22

`Séjour` contient deux clés étrangères : l'une référençant le logement, l'autre le voyageur. On peut que la valeur de `idVoyageur` (ou `codeLogement`) dans cette relation est *toujours* la valeur de l'une des clés primaire de `Voyageur` (respectivement `Logement`). Si ce n'est pas clair, vus pouvez revoir la définition des clés étrangères et méditer dessus le temps qu'il faudra.

Note : La clé étrangère `codeLogement` n'a pas la même nom que la clé primaire dont elle reprend les valeurs (`code` dans `logrement`). Au contraire, `idVoyageur`` est aussi bien le nom de la clé primaire (dans `Voyageur`) que de la clé étrangère (dans `Séjour`). Les deux situations sont parfaitement correctes et acceptables. Nous verrons comment spécifier avec SQL le rôle des attributs, indépendamment du nommage.

Connaissant un séjour, je connais donc les valeurs de clé du logement et du voyageur, et je peux trouver la description complète de ces derniers dans leur table respective. ce schéma, comme tous les bons schémas, élimine donc les redondances sans perte d'information.

La table Activité

Cette table contient les activités associées aux logements. La clé est la paire constituée de `(codeLogement, codeActivité)`.

codeLogement	codeActivité	description
pi	Voile	Pratique du dériveur et du catamaran
pi	Plongée	Baptèmes et préparation des brevets
ca	Randonnée	Sorties d'une journée en groupe
ge	Ski	Sur piste uniquement
ge	Piscine	Nage loisir non encadrée

Le schéma de cette table a une petite particularité : la clé étrangère `codeLogement` fait partie de la clé primaire. Tout se passe dans ce cas comme si on identifiait les activités relativant au logement auquel elle sont associées. Il s'agit encore une fois d'une situation normale, issue d'un de choix de conception assez courant.

Réflechissez bien à ce schéma, nous allons l'utiliser intensivement par la suite pour l'interrogation.

2.3.2 La base des films

La seconde base représente des films, leur metteur en scène, leurs acteurs. Les films sont produit dans un pays, avec une table représentant la liste des pays. De plus des internautes peuvent noter des films. Le schéma est le suivant :

- Film (**idFilm**, titre, année, genre, résumé, *idRéalisateur*, *codePays*)
- Pays (**code**, nom, langue)
- Artiste (**idArtiste**, nom, prénom, annéeNaissance)
- Rôle (**idFilm, idActeur**, nomRôle)
- Internaute (**email**, nom, prénom, région)
- Notation (**email, idFilm**, note)

Quelques choix simplifiateurs ont été faits qui demanderaient sans doute à être reconsidérés pour une base réelle. La clé étrangère `idRéalisateur` dans `Film` par exemple implique que connaissant le film, je connais son réalisateur (dépendance fonctionnelle), ce qui exclut donc d'avoir deux réalisateurs ou plus pour un même film. C'est vrai la plupart du temps, mais pas toujours.

La clé primaire de la table `Rôle` est la paire `(idFilm, idActeur)`, ce qui interdirait à un même acteur de jouer plusieurs rôles dans un même film. Là aussi, on pourrait trouver des exceptions qui rendraient ce schéma impropre à représenter tous les cas de figure. On peut donc remarquer que chaque partie de la clé de la table *Rôle* est elle-même une clé étrangère qui fait référence à une ligne dans une autre table :

- l'attribut `idFilm` fait référence à une ligne de la table *Film* (un film) ;
- l'attribut `idActeur` fait référence à une ligne de la table *Artiste* (un acteur) ;

Un même acteur peut figurer plusieurs fois dans la table `Rôle` (mais pas associé au même film), ainsi qu'un même film (mais pas associé au même acteur). Voici un exemple concis de contenu de cette base montrant les liens établis par les associations (clé primaire, clé étrangère). Commençons par la table des films.

id	titre	année	genre	idRéalisateur	codePays
20	Impitoyable	1992	Western	130	USA
21	Ennemi d'état	1998	Action	132	USA

Puis la table des artistes.

id	nom	prénom	annéeNaissance
130	Eastwood	Clint	1930
131	Hackman	Gene	1930
132	Scott	Tony	1930
133	Smith	Will	1968

En voici enfin la table des rôles, qui consiste ensentiellement en identifiants établissant des liens avec les deux tables précédentes. À vous de les décrypter pour comprendre comment toute l'information est représentée. Que peut-on dire de l'artiste 130 par exemple ? Peut-on savoir dans quels films joue Gene Hackman ? Qui a mis en scène *Impitoyable* ?

idFilm	idArtiste	nomRôle
20	130	William Munny
20	131	Little Bill
21	131	Bril
21	133	Robert Dean

La compréhension du schéma relationnel de la base sur laquelle nous travaillons est indispensable car elle sert de support à l'expression des requêtes SQL. Il est impossible d'interroger correctement une base si l'on ne sait pas comment elle est conçue, et notamment si l'on n'a pas en tête les liens définis par les clés étrangères. Comme nous le verrons, la conception d'une requête SQL s'appuie sur cette connaissance, qui peut être représenté graphique comme le montre la Fig. 2.2. Les tables y sont visualisées, et les liens entre nuplets représentés par des arêtes. On y voit par exemple qu'un rôle est lié à un film et un artiste, qu'un même film peut être lié à plusieurs rôles, qu'un artiste peut être réalisateur dans un film et acteur dans un autre, etc.

Nous nous appuierons sur cette représentation pour expliquer le raisonnement à mettre en œuvre quand on conçoit une requête SQL.

Cette base est disponible en ligne à http://deptfod.cnam.fr/bd/tp.

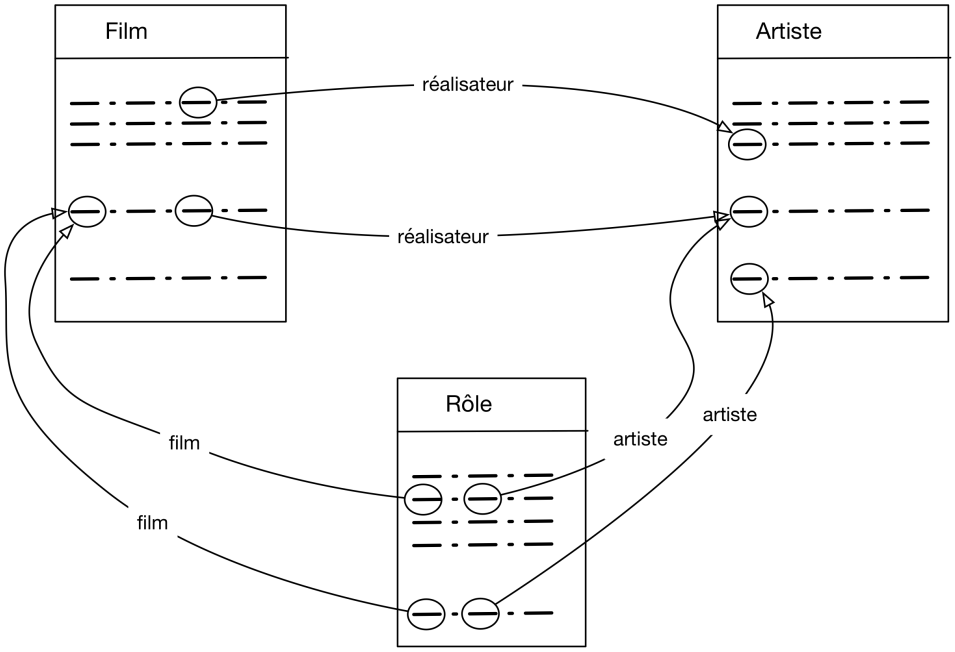

Fig. 2.2 – La base des films « vue » comme un graphe dont les arêtes sont les liens clé étrangère - clé primaire.

2.3.3 Quiz

Dans un système qui respecte la contrainte d'unicité et la contrainte d'intégrité référentielle, que pensez-vous des affirmations suivantes ?

2.4 Exercices

Exercice Ex-relationnel-1 : calculs de transitivité

Parmi les axiomes d'Armstrong figure la transitivité. Considérons la relation `R(ABCDEFGH)` qui satisfait les dépendances fonctionnelles suivantes :

- $A \to B$
- $CH \to A$
- $B \to E$
- $BD \to C$
- $A \to E$
- $EG \to H$
- $DE \to F$

Lesquelles des DFs suivantes sont également satisfaites ?

- $BFG \to AE$
- $ACG \to DH$
- $CEG \to AB$

Aide : prenez la partie gauche de la dépendance fonctionnelle et calculez par réflexivité et transitivité tous les attributs qui en sont déterminés. On appelle *clôture transitive* cet ensemble d'attributs et on le note S+, S étant l'ensemble des attributs de départ.

Exemple : prenons ACD.

— Première étape : on ajoute B car $A \rightarrow B$; E car $A \rightarrow E$
On arrive à ACDBE pour cette première étape

— On continue en ajoutant F car $DE \rightarrow F$. On regarde les parties gauches pour savoir si on peut déterminer d'autres attributs. $BD \rightarrow C$ n'apporte rien.
On arrive à ACDBEF, et on ne peut pas aller plus loin donc (ACD)+ = ACDBEF.

Exercice Ex-relationnel-1bis : clôture transitive

Soit la relation `EmpDept (enum, nom, annéeNaiss, adresse, dnum, numDirecteur)` et les DF suivantes :

— $enum \rightarrow nom, anneNaiss, adresse, dnum$
— $dnum \rightarrow numDirecteur$

Calculer enum+ et dnum+.

Exercice Ex-relationnel-2 : dépendances directes

Une dépendance est directe si elle ne peut pas être obtenue par transitivité.

— Parmi les DF de l'exercice précédent, lesquelles sont indirectes ?
— Donnez une DF indirecte sur le schéma de la relation Manuscrit vue en cours.

Exercice Ex-relationnel-3 : Augmentation et dépendances minimales

L'augmentation est un autre des axiomes d'Armstrong. Une dépendance est *minimale* si elle ne peut pas être obtenue par augmentation.

On considère la relation R(A, B, C, D) et la DF $A \rightarrow B, C, D$.

— Enumérez toutes les dépendances possibles obtenus par augmentation.

Exercice Ex-relationnel-4 : Réflexion et clés

Autre axiome d'Armstrong : la réflexion.

En l'absence de toute dépendance fonctionnelle non triviale, quelle est la clé d'une relation ?

Exercice Ex-relationnel-5 : calcul de clé

Comment calculer les clés d'une relation à partir d'un ensemble de DF ?

— Un attribut qui n'apparaît ni à gauche ni à droite d'une DF doit faire partie des clés
— Tout attribut qui apparaît à gauche mais jamais à droite d'une DF doit faire partie des clés !
— Tout attribut qui apparaît à droite mais jamais à gauche d'une DF ne peut pas faire partie d'une clé.

Donc on part des attributs qui doivent faire partie d'une clé et on vérifie qu'ils forment une clé. Si non on les *augmente* progressivement avec ceux que l'on n'arrive pas à déterminer.

Application : trouver les clés pour R (A, B, C, D, E), avec $A \to B, C \to D$
— A et C doivent faire partie de la clé
— E doit en faire partie aussi
— B et D e peuvent pas en faire partie !
— Donc toute clé *contient* (A, C, E). On constate que $A, C, E \to B, D$ donc c'est une clé. C'est la seule car elle est minimale.

Trouver les clés pour les relations suivantes :
— R(A, B, C, D, E), $A \to B; D \to E; E \to C$
— R(A, B, C, D, E), $A \to C, D; E \to C$
— R(A, B, C, D, E), $A, D \to E; E, B \to C$
— R(A, B, C, D, E), $A, B \to C; C, D \to E; D, E \to B$

Exercice Ex-relationnel-6 : argumenter sur la troisième forme normale

Soit la relation R(ABCD) et les DF `A -> B`; `C-> D` et `B -> D`

Donnez l'argument qui montre que cette relation n'est pas en troisième forme normale

---

Exercice Ex-relationnel-7 : logements et activités

Reprenons une table sur des logements et des activités.

code	nom	capacité	type	lieu	codeActivité	description
ca	Causses	45	Auberge	Cévennes	rando	Promenades faciles sur le plateau
ge	Génépi	134	Hôtel	Alpes	pisc	Nage en piscine olympique couverte
ge	Génépi	134	Hôtel	Alpes	ski	Toutes pratiques de glisse
pi	U Pinzutu	10	Gîte	Corse	rando	Trek engagés en montagne
pi	U Pinzutu	10	Gîte	Corse	pisc	Bassin à débordement face à la mer

On donne les dépendances suivantes :
— $code \to nom, capacit, type, lieu$
— $code, codeActivit \to description$

Questions :
— Quelle est la clé ?
— Cette relation est-elle en 3FN ?

Exercice Ex-relationnel-8 : analyse des DF et clés

Soit une relation `R(A, B, C, D, E, F, H, I, J)` et les dépendances fonctionnelles :
— $A, B \rightarrow C$
— $A \rightarrow D, E$
— $B \rightarrow F$
— $F \rightarrow G, H$
— $D \rightarrow I, J$
Questions : quelle est la clé, et cette relation est-elle en 3FN.

Exercice Ex-relationnel-9 : séjours

Regardons maintenant les séjours. Voici la table proposée

idVoyageur	nom	code	nomLogement	lieu	début	fin
10	Phileas Fogg	pi	U Pinzuttu	Corse	20	20
20	Nicolas Bouvier	pi	U Pinzuttu	Corse	22	23
10	Phileas Fogg	ca	Causses	Cévennes	12	15
40	R-L Stevenson	ca	Causses	Cévennes	34	36

Voici les dépendances fonctionnelles :
— $idVoyageur \rightarrow nom$
— $code \rightarrow nomLogement, lieu$
— $idVoyageur, code \rightarrow dbut, fin$
Questions : quelle est la clé, et cette relation est-elle en 3FN.

Exercice Ex-relationnel-10 : analyse des DF et clés

Soit la relation suivante, représentant des commandes d'un produit pour lequel on connaît le prix unitaire et le nombre (`nb`) d'exemplaires commandés.

```
Commande (idProduit, nomProduit, prixUnit, idClient, nomClient, date, nb, ␣
↪tauxTva, mtHT, mtTVA, mtTTC)
```

Le montant HT de la commande est le produit du nombre d'exemplaires et du prix unitaire. On sait par ailleurs que
— Le taux de TVA dépend du produit
— Le montant TTC est la somme du montant HT et du montant TVA
— Les commandes d'un même produit sont groupées quotidiennement par client
— Les attributs `id` désignent bien sûr les identifiants du produit et du client
Donner toutes les DF et trouver la clé

Exercice Ex-relationnel-11 : relations binaires

Démontrer que toute relation binaire est en troisième forme normale.

2.5 Atelier : Une étude de cas

Dans l'ensemble des exercices qui suivent, on cherche à spécifier le système d'information d'un zoo, et on suppose que l'on se trouve dans la situation suivante : une personne peu avertie (elle n'a pas suivi les enseignements du Cnam, !) a créé en tout et pour tout une seule relation dans laquelle on trouve toutes les informations. Voici le schéma de cette table.

```
Zoo (animal, nom, année_naissance, espèce, gardien,
       prénom, salaire, classe, origine, emplacement, surface)
```

Chaque ligne corrrespond à un animal auquel on attribue un nom propre, une année de naissance et une espèce (Ours, Lion, Boa, etc.). Cet animal est pris en charge par un gardien (avec prénom et salaire) et occupe un emplacement dans le zoo dont on connaît la surface. Enfin chaque espèce appartient à une classe (les mammifères, poissons, reptiles, batraciens ou oiseaux) et on considère pour simplifier qu'elle provient d'une origine unique (Afrique, Europe, etc.).

Tout cela est évidemment très approximatif. Essayons d'y mettre de l'ordre. Voici les dépendances fonctionnelles :

- animal $\rightarrow$ nom, année_naissance, espèce, emplacement.
- nom, espèce $\rightarrow$ animal.
- espèce $\rightarrow$ origine, classe.
- gardien $\rightarrow$ prénom, salaire.
- emplacement $\rightarrow$ surface, gardien.

Le but pour l'instant est d'identifier les anomalies et de trouver les clés.

2.5.1 Interprétation des dépendances

Supposons que le contenu de la table Zoo respecte les dépendances fonctionnelles ci-dessus. Répondez aux questions suivantes :

- Deux animaux peuvent-ils avoir le même nom ?
- Le nom d'un animal suffit-il pour l'identifier ?
- Peut-on avoir deux animaux avec le même nom sur le même emplacement ?
- Connaissant un animal, est-ce que je connais son origine ?
- Connaissant un animal, est-ce que je sais quel est son gardien ?
- Un gardien peut-il s'occuper de plusieurs emplacements ?
- Un emplacement peut-il être pris en charge par plusieurs gardiens ?
- Deux gardiens peuvent-ils avoir le même salaire ?

2.5.2 Recherche d'anomalies

On peut mettre n'importe quoi dans cette relation. Par exemple on pourrait y trouver le contenu de la table ci-dessous (on a simplifié le nombre de colonnes).

animal	nom	espèce	gardien	salaire	classe	origine	emplace-ment
10	Zoé	Girafe	Marcel	10 000	Poisson	Afrique	A
20	Martin	Ours	Marcel	9 000	Insecte	Europe	B
30	Martin	Girafe	Jules	12 000	Poisson	Afrique	A
20	Goupil	Renard	Marcel	10 000	Oiseau	Europe	B
40	Goupil	Renard	Jules	12 000	Insecte	Asie	A

Maintenant :

— Citer (au moins) 5 anomalies qui rendent cette table incompatible avec les dépendances fonctionnelles données précédemment.
— Citer (au moins) 2 redondances qui pourraient être évitées.

2.5.3 Calcul des clés

— Montrer que `animal` et `nom, Espèce` sont des clés de la relation `Zoo`
— Montrer que ce sont les seules clés.
— Montrer que la table n'est pas en troisième forme normale.

CHAPITRE 3

SQL, langage déclaratif

Il est courant en informatique de disposer de plusieurs langages pour résoudre un même problème. Ces langages ont leur propre syntaxe, mais surtout ils peuvent s'appuyer sur des approches de programmation très différentes. Vous avez peut-être rencontré des langages impératifs (le C), orientés-objet (Java, Python) ou fonctionnels (Camel, Erlang).

Certains langages sont plus appropriés à certaines tâches que d'autres. Il est plus facile de vérifier les propriétés d'un programme écrit en langage fonctionnel par exemple que d'un programme C. Si l'on s'en tient aux bases de données (et particulièrement pour les bases relationnelles), deux approches sont possibles : la première est *déclarative* et la seconde *procédurale*.

L'approche procédurale est assez familière : on dispose d'un ensemble d'opérations, et on décrit le calcul à effectuer par une séquence de ces opérations. Chaque opération élémentaire peut être très simple, mais la séquence à construire pour régler des problèmes complexes peut être longue et peu claire.

L'approche déclarative est beaucoup plus simple conceptuellement : elle consiste à décrire les propriétés du point d'arrivée (le résultat) en fonction de celles du point de départ (les données de la base, dans notre cas). La description de ces propriétés se fait classiquement par des formules logiques qui indiquent comment l'existence d'un fait f_1 au départ implique l'existence d'un fait f_2 à l'arrivée.

Cela peut paraître abstrait, et de fait ça l'est puisqu'aucun calcul n'est spécifié. On s'appuie simplement sur le fait que l'informatique *sait* effectuer des calculs spécifiés par des formules logiques (dans le cas particulier des bases de données en tout cas) apparemment indépendantes de tout processus calculatoire. Il se trouve que SQL *est* un langage déclaratif, et qu'il l'était même exclusivement dans sa version initiale.

Note : Il existe de très bonnes raisons pour privilégier le caractère déclaratif des langages de requêtes, liées à l'indépendance entre le niveau logique et le niveau physique donc nous avons déjà parlé, et à l'opportunité que cette indépendance laisse au SGBD pour déterminer la meilleure manière d'évaluer une requête. Cela n'est possible que si l'expression de cette requête est assez abstraite pour n'imposer aucun choix de calcul à

priori.

Avec SQL, on ne dit rien sur la manière dont le résultat doit être calculé : c'est le problème du SGBD, qui sait d'ailleurs trouver la solution bien mieux que nous puisqu'on ne connaît pas l'organisation des données. On se contente avec SQL d'énoncer les propriétés de la relation de sortie en fonction des propriétés de la base en entrée. Pour bien utiliser SQL, et surtout bien comprendre la *signification* de ce que l'on exprime, il faut donc maîtriser l'expression de formules logiques et connaitre les mécanismes d'inférences des valeurs de vérité.

On rencontre parfois l'argument que SQL est, à l'inverse d'un langage de programmation, accessible à un non-initié, car il est proche de la manière dont on exprimerait naturellement une recherche. Ce n'est vrai que si on sait *formuler* cette dernière de manière rigoureuse, et c'est exactement ce que nous allons apprendre dans ce chapitre.

SQL est-il *totalement* déclaratif ?

Au fil des années et des normes successives, SQL s'est étendu pour incorporer un autre langage relationnel, l'algèbre, que nous étudierons dans le prochain chapitre. Est-ce à dire que la forme déclarative n'était pas suffisante ? Non : tous ces ajouts sont redondants et auraient pu être omis sans affecter l'expressivité du langage.

On se retrouve à l'heure actuelle avec un langage très riche dans lequel on peut exprimer des requêtes de manière soit déclarative, soit procédurale, soit par un mélange des deux. Cela ne contribue pas forcément à la facilité d'apprentissage, et introduit une certaine confusion sur la portée de telle ou telle formulation, et sa possible équivalence avec une autre.

En présentant successivement les deux approches, et en montrant ensuite comment elles sont parfaitement équivalentes l'une à l'autre, ce cours a choisi de tenter de clarifier la situation.

3.1 S1 : Un peu de logique

Supports complémentaires :

— Diapositives: notions de logique
— Vidéo sur les notions de logique

La logique est l'art de raisonner, autrement dit de construire des argumentations rigoureuses permettant d'induire ou déduire de nouveaux faits à partir de faits existants (ou considérés comme tels). La logique mathématique est la partie de la logique qui présente les règles de raisonnement de manière formelle. C'est une branche importante des mathématiques, qui s'est fortement développée au début du XXe siècle, et constitue un fondement majeur de la science informatique.

Commençons par effectuer un rappel des quelques éléments de logique formelle qui sont indispensables pour formuler et interpréter les requêtes SQL. Ce qui suit n'est qu'une très brève (et assez simplifiée) introduction au sujet : il faut recourir à des textes spécialisés si vous voulez aller plus loin. Pour une passionante intro-

duction historico-scientifique, je vous recommande d'ailleurs la bande dessinée (mais oui) *Logicomix*, parue chez Vuivert en 2009.

Important : Ceux qui pensent maîtriser le sujet peuvent sauter cette session.

La partie la plus simple de la logique formelle est le calcul propositionnel, par lequel nous commençons. SQL est construit sur une forme plus élaborée, impliquant des *prédicats*, des *collections* et des *quantificateurs*, notions brièvement présentées ensuite dans une optique « bases de données ».

3.1.1 Le calcul propositionnel

Une *proposition* est un énoncé auquel on peut attacher une valeur de vérité : vrai (V) ou faux (F). Des énoncés comme « Ce livre parle d'informatique » ou « Cette musique est de Mozart » sont des propositions. Une question comme « Qui a écrit ce texte ? » n'est pas une proposition.

Le calcul propositionnel décrit la manière dont on peut combiner des propositions pour former des *formules* (propositionnelles) et attacher des valeurs de vérité à ces formules. Les propositions peuvent être combinées grâce à des *connecteurs logiques*. Il en faut au moins deux, mais on en considère en général trois.

— la conjonction, notée $\wedge$
— la disjonction, notée $\vee$
— la négation, notée $\neg$

On note classiquement les propositions par des lettres en minuscules, p, q, r. La table ci-dessous donne les valeurs de vérités pour les formules obtenues à l'aide des trois connecteurs logiques, en fonction des valeurs de vérité de p et q.

Tableau 3.1 – Valeurs de vérité pour les connecteurs logiques

p	q	$p \wedge q$	$p \vee q$	$\neg p$
V	V	V	V	F
V	F	F	V	F
F	V	F	V	V
F	F	F	F	V

Les formules créées par connecteurs logiques à partir de propositions ont elles-mêmes des valeurs de vérité, et on peut les combiner à leur tour. Généralement, si F_1 et F_2 sont des formules, alors $F_1 \wedge F_2$, $F_1 \vee F_2$ et $\neg F_1$ sont aussi des formules. On crée ainsi des *arbres* dont les feuilles sont des propositions et les nœuds internes des connecteurs.

Pour représenter l'arbre dans le codage de la formule, on utilise des parenthèses et on évite ainsi toute ambiguité. La formule $p \wedge q$ peut ainsi être combinée à r selon la syntaxe.

$$(p \wedge q) \vee r$$

La valeur de vérité de la formule obtenue s'obtient par application récursive des règles de la table. Si, par exemple, p, q, r ont respectivement pour valeurs V, V et F, la formule ci-dessus s'interprète ainsi :

— $(p \wedge q)$ vaut $(V \wedge V)$, donc V
— $(p \wedge q) \vee r$ vaut $V \vee r$, qui vaut $V \vee F$, donc V

Deux formules sont *équivalentes* si elles ont les mêmes valeurs de vérité quelles que soient les valeurs initiales des propositions. Les équivalences les plus courantes sont très utiles à connaître. En notant F, F_1, F_2 trois formules quelconques, on a :

— $\neg(\neg F)$ est équivalente à F
— $F_1 \wedge F_2$ est équivalente à $F_2 \wedge F_1$ (commutativité)
— $F_1 \vee F_2$ est équivalente à $F_2 \vee F_1$ (commutativité)
— $F \wedge (F_1 \wedge F_2)$ est équivalente à $(F \wedge F_1) \wedge F_2$ (associativité)
— $F \vee (F_1 \vee F_2)$ est équivalente à $(F \vee F_1) \vee F_2$ (associativité)
— $F \vee (F_1 \wedge F_2)$ est équivalente à $(F \vee F_1) \wedge (F \vee F_2)$ (distribution)
— $F \wedge (F_1 \vee F_2)$ est équivalente à $(F \wedge F_1) \vee (F \wedge F_2)$ (distribution)
— $\neg(F_1 \wedge F_2)$ est équivalente à $(\neg F_1) \vee (\neg F_2)$ (loi DeMorgan)
— $\neg(F_1 \vee F_2)$ est équivalente à $(\neg F_1) \wedge (\neg F_2)$ (loi DeMorgan)

Une *tautologie* est une formule qui est toujours vraie. La tautologie la plus évidente est

$$F \vee \neg F$$

Une *contradiction* est une formule qui est toujours fausse. La contradiction la plus évidente est

$$F \wedge \neg F$$

Vérifiez que ces notions sont claires pour vous : il est bien difficile d'écrire correctement du SQL si on ne les maîtrise pas.

Note : Un connecteur très intéressant est l'implication, noté $\rightarrow$. L'implication ne fait pas partie de connecteurs primaires car $p \rightarrow q$ est équivalent à $\neg p \vee q$. Nous y revenons dans les exercices. Et, oui, l'implication a un lien avec les dépendances fonctionnelles : nous y revenons aussi !

3.1.2 Prédicats

Une faiblesse de la logique propositionnelle est qu'elle ne considère que des énoncés « bruts », non décomposables. Si je considère les énoncés « Mozart a composé Don Giovanni », « Mozart a composé Cosi fan tutte », et « Bach a composé la Messe en si », la logique propositionnelle ne permet pas de distinguer qu'ils déclarent le même type de propriété (le fait de composer une œuvre) liant des entités (Mozart, Bach, leurs œuvres). Il est impossible par exemple en calcul propositionnel d'identifier que deux des propositions parlent de la même entité, Mozart.

Les *prédicats* sont des extensions des propositions qui énoncent des propriétés liant des objets. Un prédicat est de la forme $P(X_1, X_2, \cdots, X_n)$, avec $n \geq 0$, P étant le nom du prédicat, et les X_i désignant les entités liés par la propriété. On peut ainsi définir un prédicat $Compose(X, Y)$ énonçant une relation de type *X a composé Y* entre l'entité représentée par *X* et celle représentée par *Y*.

Avec un prédicat, il est possible de donner un ensemble d'énoncés ayant tous la même structure. Appelons ces énoncés des *nuplets* pour adopter une terminologie « bases de données » (un logicien parlera plutôt d'atôme, ou de fait). Les trois propositions précédentes deviennent donc les trois nuplets suivants :

```
Compose (Mozart, Don Giovanni)
Compose (Mozart, Cosi fan tutte)
Compose (Bach, Messe en si)
```

Cette fois, contrairement au langage propositionnel, on désigne explicitement les entités : compositeurs (dont Mozart, qui apparaît deux fois) et œuvres. On obtient une collection de *nuplets* qui remplacent avantageusement les propositions grâce à leur structure plus riche.

Il existe virtuellement une infinité de nuplets énoncables avec un prédicat. Certains sont faux, d'autres vrais. Comment les distingue-t-on ? Tout dépend du contexte interprétatif.

Dans un contexte arithmétique par exemple, les prédicats courants sont l'égalité, l'inégalité (stricte ou large) et leur négation. Le prédicat d'égalité s'applique à deux valeurs numériques et s'écrit $=\ (x, y)$. L'interprétation de ces prédicats (dans un contexte arithmétique encore une fois) est celle que nous connaissons « naturellement ». On sait par exemple que $\geq (2, 1)$ est vrai, et que $\geq (1, 2)$ est faux.

Quand on modélise le monde réel, les nuplets vrais doivent le plus souvent être énoncés explicitement comme, dans l'exemple ci-dessus, les compositeurs et leurs œuvres. Une base de données n'est rien d'autre que l'ensemble des nuplets considérés comme vrais pour des prédicats applicatifs, tous les autres étant considérés comme faux.

Un système pourra nous dire que le nuplet suivant est faux (il n'est pas dans la base) :

```
Compose (Bach, Don Giovanni)
```

Alors que le nuplet suivant est vrai (il appartient à la base) :

```
Compose (Mozart, Don Giovanni)
```

Une réponse Vrai/Faux n'est pas forcément très utile. Nous restons pour l'instant dans un système assez restreint où tous les nuplets font référence à des entités connues. De tels nuplets sont dits *fermés*. Mais on peut également manipuler des nuplets dits *ouverts* dans lesquels certains objets sont inconnus, et remplacés par des variables habituellement dénotés x, y, z. On obtient un langage beaucoup plus puissant.

Dans le nuplet ouvert suivant, le nom du compositeur est remplacé par une variable.

$$Compose(x, \text{DonGiovanni})$$

Intuitivement, ce nuplet ouvert représente concisément tous les nuplets fermés exprimant qu'un musicien x a composé une œuvre intitulée Don Giovanni. En affectant à x toutes les valeurs possibles (une variable est supposée couvrir un domaine de valeurs), on énumère tous les nuplets de ce type. La plupart sont faux (ceux qui ne sont pas dans la base), certains sont vrais.

Interroger une base relationnelle, c'est simplement demander au système les valeurs de x pour lesquelles $Compose(x, DonGiovanni)$ est vrai. La réponse est probablement `Mozart`.

3.1.3 Collections et quantificateurs

L'ensemble des nuplets vrais d'un prédicat constitue une *collection*. Jusqu'à présent nous avons évalué les valeurs de vérité au niveau de chaque nuplet individuel, mais on peut également le faire sur l'ensemble de la collection grâce aux quantificateurs *existentiel* et *universel*.

— Le quantificateur existentiel. $\exists x P(x)$ est vrai s'il existe *au moins* une valeur de x pour laquelle $P(x)$ est vraie.
— Le quantificateur universel. $\forall x P(x)$ est vrai si $P(x)$ est vraie pour toutes les valeurs de x.

Note : Le quantificateur existentiel serait suffisant puisqu'il est possible d'exprimer la quantification universelle avec deux négations. Une propriété P est *toujours* vraie s'il *n'existe pas* de cas où est *n'est pas* vraie. SQL ne connaît d'ailleurs que le `exists` : voir plus loin.

On peut donc définir la forme complète des formules de la manière suivante :

Définition : Syntaxe des formules

- Un nuplet (ouvert ou fermé) $P(a_1, a_2, \cdots, a_n)$ est une formule
- Si F_1 et F_2 sont deux formules, $F_1 \wedge F_2$, $F_1 \vee F_2$ et $\neg F_1$ sont des formules.
- Si F est une formule et si x est une variable, alors $\exists x F$ et $\forall x F$ sont des formules.

Les notions d'« ouvert » et de « fermé » se généralisent au niveau des formules : une formule est *ouverte* si elle contient des variables qui ne sont liées par aucun quantificateur. On les appelle les *variables libres*. Les formules ouvertes sont celles qui nous intéressent en base de données, car elles reviennent à poser la question suivante : *quelles sont les valeurs des variables libres qui satisfont (rendent vraie) la formule* ?

Reprenons l'un des exemples précédents

$$Compose(x, \text{Don Giovanni})$$

Cette formule est ouverte, avec une seule variable libre ; x. Les valeurs qui satisfont cette formule sont les noms des compositeurs qui ont écrit *Don Giovanni*.

Voici une autre formule dans laquelle le second composant est une variable dite « anonyme », notée _.

$$Compose(x, _)$$

Variable anonyme

Les variables anonymes sont celles dont la valeur ne nous intéresse pas et auxquelles on ne se donne donc même pas la peine de donner un nom. Ecrire un nuplet $P(_, x, _, _)$ est donc une facilité d'écriture pour ne pas avoir à nommer trois variables qui ne servent à rien. La notation complète serait de la forme $P(x_1, x, x_2, x_3)$.

La seule variable libre est x, et les valeurs de x qui satisfont la formule sont l'ensemble des noms de compositeur. De fait, une formule F avec des variables libres $x_1, x_2, \cdots, x_n$ définit un prédicat $R(x_1, x_2, \cdots, x_n)$. L'ensemble des nuplets vrais de R est l'ensemble des nuplets qui satisfont F. Pour reprendre notre exemple, on pourrait définir le prédicat *Compositeur* de la manière suivante :

$$Compositeur(x) \leftarrow Compose(x, _)$$

Ce qui se lit : « x est un compositeur s'il existe une valeur de y telle que (x, y) est un nuplet vrai de *Compose*.

Un schéma de base de données peut être vu comme la déclaration d'un ensemble de prédicats. Reprenons un exemple déjà rencontré, celui des manuscrits évalués par des experts.

- Expert (id_expert, nom)
- Manuscrit (id_manuscrit, auteur, titre, id_expert, commentaire)

Ces prédicats énoncent des propriétés. Le premier nous dit que l'expert nommé `nom` a pour identifiant `id_expert`. Le second nous dit que le manuscrit identifié par `id_manuscrit` s'intitule `titre`, a été rédigé par `auteur` et évalué part l'expert identifié par `id_expert` qui a ajouté un `commentaire`.

Voici quelques formules sur ces prédicats. La première est vraie pour toutes les valeurs de x égales à l'identifiant d'un expert nommé Serge.

$$Expert(x, \text{'Serge'})$$

La seconde est vraie pour toutes les valeurs de t titre du manuscrit d'un auteur nommé Proust.

$$Manuscrit(_, \text{'Proust'}, t, _)$$

Enfin, la troisième est vraie pour toutes les valeurs de t , x et n telles que t est le titre du manuscrit d'un auteur nommé Proust, évalué par un expert identifié par x et nommé n.

$$Manuscrit(_, \text{'Proust'}, t, x, _) \wedge Expert(x, n)$$

Notez que la variable x est utilisée à la fois dans `Manuscrit` et dans `Expert`. Cela contraint une valeur de x à être *à la fois* un identifiant d'un expert dans `Expert`, et la valeur de la clé étrangère de cet expert dans `Manuscrit` Autrement dit l'énoncé de cette formule lie un manuscrit à l'expert qui l'a évalué. Ce mécanisme de lien par partage de valeur, nommé *jointure* est fondamental dans l'interrogation de bases relationnelles ; nous aurons l'occasion d'y revenir longuement.

Voici une dernière formule qui illustre l'utilisation des quantificateurs.

$$Expert(x, n) \wedge \exists Manuscrit(_, \text{'Proust'}, _, x, _)$$

Cette formule s'énonce ainsi : tous les experts n qui ont évalué *au moins* un manuscrit d'un auteur nommé Proust. Notez encore une fois la présence de l'identifiant de l'expert dans les deux nuplets libres, sur Expert et Manuscrit.

3.1.4 Logique et bases de données

Ce qui précède peut sembler inutilement conceptuel ou compliqué, surtout au vu de la simplicité des exemples donnés jusqu'à présent. Il faut bien réaliser que ces exemples ne sont qu'une illustration d'une méthode d'interrogation très générale dans laquelle on demande au système de nous fournir, à partir des nuplets de la base (ceux considérés comme vrais), toutes les informations qui satisfont une propriété logique. Cette propriété s'exprime dans le langage de la logique formelle, ce qui offre des avantages décisifs :

- la signification d'une formule et précise, non ambiguë ;
- il existe des algorithmes efficaces pour évaluer la valeur de vérité d'une formule ;
- le langage est robuste, universellement connu et adopté, ce qui permet d'obtenir un mode d'interrogation *normalisé*.
- enfin, ce langage est totalement *déclaratif* : exprimer une formule ne donne aucune indication sur la manière dont le système doit trouver le résultat.

SQL, dans sa forme déclarative, qui est la forme d'origine, est un langage concret pour écrire des formules logiques que le SGBD se charge d'interpréter et de calculer. Reprenons la formule :

$$Compose(x, \text{Don Giovanni})$$

Voici la requête SQL correspondante.

```
select compositeur
from Compose
where oeuvre='Don Giovanni'
```

Et voici la forme SQL des deux dernières formules sur les experts (avec jointure)

```
select titre, nom
from Expert, Manuscrit
where Expert.id_expert = Manuscrit.id_expert
and auteur = 'Proust'
```

```
select nom
from Expert
where exists (select *
              from Manuscrit
              where Expert.id_expert = Manuscrit.id_expert
              and auteur = 'Proust')
```

Maîtriser l'expression des formules et, surtout, comprendre leur signification précisément, est donc une condition pour utiliser SQL correctement.

3.1.5 Quiz

3.2 S2 : SQL conjonctif

Supports complémentaires :

— Diapositives: SQL conjonctif

— Vidéo sur la première partie de SQL

Cette session présente le langage SQL dans sa version déclarative, chaque requête s'interprétant par une formule logique. La base de données est constituée d'un ensemble de relations vues comme des prédicats. Ces relations contiennent des nuplets (fermés, sans variable).

Note : Prédicats ou relations ?

Un prédicat énonce une propriété liant des objets, et est donc synonyme de *relation* au sens mathématique du terme. Les deux termes peuvent être utilisés de manière interchangeable.

Pour illustrer les requêtes et leur interprétation, nous prenons la base des voyageurs présentée dans le chapitre *Le modèle relationnel*. Vous pouvez expérimenter toutes les requêtes présentées (et d'autres) directement sur note site http://deptfod.cnam.fr/bd/tp. Voir également l'atelier SQL proposé en fin de chapitre.

Cette session se limite à la partie dite « conjonctive » de SQL, celle où toutes les requêtes peuvent s'exprimer sans négation. La prochaine session complètera le langage.

3.2.1 Requête mono-variable

Dans les requêtes relationnelles, les variables ne désignent pas des valeurs individuelles, mais des nuplets libres. Une variable-nuplet t a donc des composants $a_1, a_2, \ldots a_n$ que l'on désigne par $t.a_1, t.a_2, \cdots, t.a_n$. Par souci de simplicité, on nomme souvent les variables comme les attributs du schéma, mais ce n'est pas une obligation.

Commençons par étudier les requêtes utilisant une seule variable. Leur forme générale est

```
select [distinct] t.a1, t.a2, ..., t.an
from T as t
where <condition>
```

Ce « bloc » SQL comprend trois clauses : le `from` définit la variable libre et ce que nous appellerons la *portée* de cette variable, le `where` exprime les conditions sur la variable libre, enfin le `select`, accompagné du mot-clé optionnel `distinct`, construit le nuplet constituant le résultat. Cette requête correspond à la formule :

$$\{t.a_1, t.a_2, \cdots, t.a_n | T(t) \wedge F_{cond}(t)\}$$

L'interprétation est la suivante : je veux constituer tous les nuplets *fermés* $(t.a_1, t.a_2, \cdots, t.a_n)$ dont les valeurs satisfont la formule $T(t) \wedge F_{cond}$. Cette formule comprend toujours deux parties :

- La première, $T(t)$ indique que la variable t est un nuplet de la relation T. Autrement dit $T(t)$ est vraie si $t \in T$. Nous appelons donc cette partie la *portée*.
- La seconde, $F_{cond}(t)$, est une formule logique sur t, que nous appellons la *condition*.

Important : La portée définit les variables libres de la formule, celles pour lesquelles on va chercher l'affectation qui satisfait la condition $F_{cond}(t)$, et à partir desquelles on va construire le nuplet-résultat. Reportez-vous à la session précédente pour la notion de variable libre dans une formule et leur rôle dans un système d'interrogation.

À propos du `distinct`

Une relation ne contient pas de doublon. La présence de doublons (deux unités d'information indistinguables l'une de l'autre) dans un système d'information est une anomalie. Pour prendre quelques exemples applicatifs, on ne veut pas envoyer deux fois le même message, on ne veut pas produire deux fois la même facture, on ne veut pas afficher deux fois le même document, etc. Vous pouvez vérifier que votre moteur de recherche préféré applique ce principe.

Les relations de la base sont en première forme normale, et la présence de doublons est évitée par la présence d'au moins une clé. Qu'en est-il des relations calculées, autrement dit le résultat des requêtes ? Supposons que l'on souhaite connaître tous les types de logements. Voici la requête SQL sans `distinct` :

```
select type
from Logement
```

On obtient une relation avec deux nuplets identiques.

type
Gîte
Hôtel
Auberge
Hôtel

Sans `distinct`, SQL peut produire des relations avec doublons. Du point de vue logique, cela montre simplement que l'on a établi le même fait de deux manières différentes, mais cela ne sert à rien d'afficher ce fait deux fois (ou plus). Si on ajoute `distinct`

```
select distinct type
from Logement
```

on obtient

type
Gîte
Hôtel
Auberge

Pourquoi SQL n'élimine-t-il pas systématiquement les doublons ? En premier lieu parce que cette élimination implique un algorithme potentiellement coûteux si la relation en entrée est très grande. Il faut en effet effectuer un tri suivi d'une élimination des nuplets identiques. Sur des petites relations, la différence en temps d'exécution est indiscernable, mais elle peut devenir significative quand on a des centaines de milliers de nuplets ou plus. Les concepteurs du langage SQL ont fait le choix, par défaut, d'éviter d'appliquer cet algorithme, ce qui revient à accepter de produire éventuellement des doublons.

Une seconde raison pour ne pas appliquer systématiquement l'algorithme d'élimination de doublons est que certaines requêtes, par construction, produisent un résultat sans doublons. Voici un exemple très simple

```
select code, type
from Logement
```

Inutile dans ces cas-là d'utiliser `distinct` (voyez-vous pourquoi ?). En d'autres termes : SQL nous laisse la charge de décider quand une requête risque de produire des doublons, et si nous souhaitons les éliminer. *Dans tout ce cours nous utilisons* `distinct` *chaque fois que c'est nécessaire pour toujours obtenir un résultat en première forme normale, sans doublon.*

Comment savoir si une requête risque de produire des doublons ?

C'est une bonne question. L'exemple donné ci-dessus nous donne une piste : il nous faut des dépendances fonctionnelles dans le résultat ! Voir l'atelier en fin de chapitre.

Il est par ailleurs très utile, quand on exprime une requête, de réfléchir à la possibilité qu'elle produise ou non des doublons et donc à la nécessité d'utiliser `distinct`. Si une requête produit potentiellement des doublons, il est sans doute pertinent de se demander quel est le sens du résultat obtenu.

Exemples

Voici une première requête concrète sur notre base. On veut le nom et le type des logements corses.

```
select t.code, t.nom, t.type
from Logement as t
where t.lieu = 'Corse'
```

Note : Pour distinguer les chaînes de caractères des noms d'attribut, on les encadre par des apostrophes simples.

Elle correspond à la formule :

$$\{t.code, t.nom, t.type | Logement(t) \wedge t.lieu = \text{'Corse'}\}$$

Note : SQL permet, quand c'est possible, quelques légères simplifications syntaxiques. La forme simplfiée de la requête précédente est donnée ci-dessous.

```
select code, nom, type
from Logement
where lieu = 'Corse'
```

On peut donc omettre de spécifier le nom de la variable quand il n'y a pas d'ambiguité, notamment l'interprétation du nom des champs.

Elle s'interprète de la manière suivante : on cherche les affectations d'une variable t parmi les nuplets de la relation `Logement`, telle que `t.lieu` ait pour valeur « Corse ».

De cette interprétation, assez évidente pour l'instant, il faut retenir qu'une table mentionnée dans le `from` de SQL définit en fait une variable dont la portée est la table (ici, `Logement`). Parmi toutes les affectations possibles de cette variable, on ne conserve que celles qui satisfont la condition exprimée par le reste de la formule.

Le système d'évaluation peut donc considérer que t est affectée à n'importe lequel des nuplets de la table, et évaluer si cette affectation satisfait la condition. Dans la table ci-dessous, la croix indique à quel nuplet t est affectée. Ici, la condition n'est clairement pas satisfaite.

t	code	nom	capacité	type	lieu
	pi	U Pinzutu	10	Gîte	Corse
	ta	Tabriz	34	Hôtel	Bretagne
X	ca	Causses	45	Auberge	Cévennes
	ge	Génépi	134	Hôtel	Alpes

En revanche, quand l'affectation est faite comme indiquée ci-dessous, la condition est satisfaite. L'affectation de la variable t satisfait alors l'ensemble de la formule et sert à construire le nuplet-résultat.

t	code	nom	capa-cité	type	lieu
X	pi	U Pinzutu	10	Gîte	Corse
	ta	Tabriz	34	Hôtel	Bretagne
	ca	Causses	45	Auberge	Cévennes
	ge	Génépi	134	Hôtel	Alpes

À partir de là, il suffit de savoir exprimer une formule pour spécifier correctement une requête SQL.

Voici quelques exemples. Cherchons d'abord quels hôtels sont dans les Alpes. La requête SQL est :

```
select t.code, t.nom
from Logement as t
where t.type = 'Hôtel' and t.lieu = 'Alpes'
```

Elle correspond à la requête logique

$$\{t.code, t.nom | Logement(t) \wedge t.type = \text{'Hôtel'} \wedge t.lieu = \text{'Alpes'}\}$$

La condition à satisfaire pour un nuplet de la relation `Logement` est $t.type = $ 'Hôtel' $\wedge\, t.lieu = $ 'Alpes'. C'est seulement le cas pour le dernier nuplet. Cherchons maintenant les hôtels qui, soit sont en Bretagne, soit ont au moins 100 chambres. La version SQL :

```
select t.code, t.nom
from Logement as t
where t.type = 'Hôtel' and (t.lieu = 'Alpes' or t.capacité >= 100)
```

Et sa version logique :

$$\{t.code, t.nom | Logement(t) \wedge t.type = \text{'Hôtel'} \wedge (t.lieu = \text{'Alpes'} \vee t.capacit \geq 100)\}$$

3.2.2 Requêtes multi-variables

Voypns maintenant le cas général où on s'autorise à utiliser plusieurs variables. Pour simplifier la notation, nous allons étudier les requêtes avec exactement deux variables. Il est facile ensuite de généraliser.

Leur forme est

$$\{t_1.a_1, \cdots, t_1.a_n, t_2.b_1, \cdots, t_2.b_m | T_1(t_1) \wedge T_2(t_2) \wedge F_{cond}(t_1, t_2)\}$$

On retrouve dans la formule les deux parties : la portée indique les relations respectives qui servent de domaine d'affectation pour t_1 et t_2 ; la condition est une formule avec t_1 et t_2 comme variables libres.

La transcription en SQL est presque littérale.

```
select [distinct] t1.a1, ..., t1.an, t2.b1, ..., t2.bm
from T1 as t1, T2 as t2
where <condition>
```

L'interprétation est exactement la même que pour les requêtes mono-variables, légèrement généralisée : *parmi toutes les affectations possibles des variables, on ne conserve que celles qui satisfont la condition exprimée par le reste de la formule*.

Il n'y a rien de plus à comprendre. Il suffit de considérer toutes les affectations possibles de t_1 et t_2 et de ne garder que celles pour lesquelles la formule de condition est satisfaite.

Voici quelques exemples. On veut les noms des logements où on peut pratiquer le ski. Nous avons besoin de deux variables :

— la première s'affecte aux nuplets de la table `Activité` ; on ne veut que ceux dont le code est `Ski`.
— la seconde s'affecte aux nuplets de la table `Logement`

Enfin, une condition doit lier les deux variables : on veut qu'elles soient relatives au même logement, et donc que le code logement soit identique. Voici la formule, suivie de la requête SQL.

$$\{l.code, l.nom | Logement(l) \wedge Activit(a) \wedge l.code = a.codeLogement \wedge a.codeActivit = \text{'Ski'}\}$$

Remarquons au passage que le nom que l'on donne aux variables n'a aucune importance. Nous utilisons `l` pour le logement, `a` pour l'activité.

```
select l.code, l.nom
from Logement as l, Activité as a
where l.code = a.codeLogement
and   a.codeActivité = 'Ski'
```

Les seules affectations de l et a satisfaisant la formule sont marquées par des croix dans les tables ci-dessous (les champs concernés ont de plus été mis en gras). Prenez, si nécessaire, le temps de bien comprendre que d'une part la formule de condition est bien satisfaite, et d'autre part qu'il n'y a pas d'autre solution possible.

l	code	nom	capa-cité	type	lieu
	pi	U Pinzutu	10	Gîte	Corse
	ta	Tabriz	34	Hôtel	Bretagne
	ca	Causses	45	Auberge	Cévennes
X	**ge**	Génépi	134	Hôtel	Alpes

a	codeLoge-ment	codeActivité	description
	pi	Voile	Pratique du dériveur et du catamaran
	pi	Plongée	Baptèmes et préparation des brevets
	ca	Randonnée	Sorties d'une journée en groupe
X	**ge**	**Ski**	Sur piste uniquement
	ge	Piscine	Nage loisir non encadrée

A partir de ces deux affectations, on construit le résultat.

code	nom
ge	Génépi

Pour maîtriser cette partie de SQL (sans doute la plus couramment utilisée), il faut bien comprendre le mécanisme mis en œuvre. Pour construire un nuplet du résultat, nous avons besoin de 1, 2 ou plus nuplets provenant de la base. Il faut identifier ces nuplets, les conditions qu'ils doivent satisfaire, et les valeurs qu'ils partagent. Ici :

— nous avons besoin d'un nuplet de la relation `Activité`, tel que le code soit `Ski` ;
— nous avons besoin d'un nuplet de la relation `Logement`, puisque nous souhaitons obtenir le nom du logement en sortie ;
— enfin ces nuplets doivent être relatifs au même logement, et partager donc la même valeur sur l'attribut qui identifie ce logement, respectivement `code` dans `Logement` et `codeLogement` dans `Activité`.

Ce raisonnement est très général et permet d'exprimer des requêtes SQL puissantes. Les seules conditions sont de formuler rigoureusement la requête et de comprendre le schéma de la base.

Prenons un autre exemple montrant que l'on peut utiliser la même portée pour des variables différentes. On veut obtenir les paires de logements qui sont du même type. Puisqu'il nous faut deux logements, nous avons besoin de deux variables, ayant chacune pour portée la table `Logement`. Ces deux variables doivent partager la même valeur pour l'attribut `type`. Voici la formule :

$$\{l_1.nom, l_2.nom | Logement(l_1) \wedge Logement(l_2) \wedge l_1.type = l_2.type\}$$

Les deux variables ont été nommées respectivement l_1 et l_2. La syntaxe SQL est donnée ci-dessous.

```
select distinct l1.nom as nom1, l2.nom as nom2
from Logement as l1, Logement as l2
where l1.type = l2.type
```

Note : Dans la syntaxe SQL, il faut résoudre les ambiguités éventuelles sur les noms d'attributs avec `as`. Ici, on a nommé le nom du premier logement `nom1` et celui du second `nom2` pour obtenir en sortie une relation de schéma `(nom1, nom2)`.

Il existe plusieurs affectations de `l1` et `l2` pour lesquelles la formule est satisfaite. La première est donnée ci-dessous : `l1` est affectée à la seconde ligne et `l2` à la quatrième.

l1	l2	code	nom	capa-cité	type	lieu
		pi	U Pinzutu	10	Gîte	Corse
X		ta	Tabriz	34	Hôtel	Bretagne
		ca	Causses	45	Auberge	Cévennes
	X	ge	Génépi	134	Hôtel	Alpes

Mais la formule est également satisfaite si on inverse les affectations : `l1` est à la quatrième ligne et `l2` à la seconde.

l1	l2	code	nom	capa-cité	type	lieu
		pi	U Pinzutu	10	Gîte	Corse
	X	ta	Tabriz	34	Hôtel	Bretagne
		ca	Causses	45	Auberge	Cévennes
X		ge	Génépi	134	Hôtel	Alpes

Et, surprise, elle est également satisfaite si les deux variables sont affectées au *même* nuplet.

l1	l2	code	nom	capa-cité	type	lieu
X	X	pi	U Pinzutu	10	Gîte	Corse
		ta	Tabriz	34	Hôtel	Bretagne
		ca	Causses	45	Auberge	Cévennes
		ge	Génépi	134	Hôtel	Alpes

Pour éviter les inversions et auto-égalités, on peut ajouter une condition :

```
select distinct l1.nom as nom1, l2.nom as nom2
from Logement as l1, Logement as l2
where l1.type = l2.type
and l1.nom < l2.nom
```

Le résultat de cette requête est alors :

nom1	nom2
Génépi	Tabriz

Interprétation d'une requête SQL

En résumé, *quelle que soit sa complexité*, l'interprétation d'une requête SQL peut *toujours* se faire de la manière suivante.

— Chaque variable du `from` peut être affectée à tous les nuplets de sa portée.
— Le `where` définit une condition sur ces variables : seules les affectations satisfaisant cette condition sont conservées
— Le nuplet résultat est construit à partir de ces affectations

---

Remarquez que ce mode d'interrogation n'indique en aucune manière, même de très loin, comment le résultat est calculé. On est (pour insister) dans une approche purement *déclarative* où le système est totalement libre de déterminer la méthode la plus efficace.

3.2.3 Quiz

3.3 S3 : Quantificateurs et négation

Supports complémentaires :
- Diapositives: SQL: quantificateurs et négation
- Vidéo sur les quantificateurs et la négation dans SQL

Jusqu'à présent les seules variables que nous utilisons sont des variables libres de la formule, définies dans la clause `from` de la syntaxe SQL. Nous n'avons pas encore rencontré de variable liée parce que nous n'avons pas utilisé les quantificateurs.

SQL propose uniquement le quantificateur existentiel. Le quantificateur universel peut être obtenu en le combinant avec la négation. Rappelons que les quantificateurs servent à exprimer des conditions sur l'ensemble d'une relation (qui peut être une relation en base, ou une relation calculée). Ils sont particulièrement utiles pour les requêtes qui comportent des *négations* (« je *ne* veux *pas* des objets qui ont telle ou telle propriété dans mon résultat »).

3.3.1 Le quantificateur `exists`

Reprenons simplement la requête qui demande les logements où l'on peut faire du ski. La formule donnée précédemment est la suivante :

$$\{l.nom | Logement(l) \wedge Activit(a) \wedge l.code = a.codeLogement \wedge a.codeActivit = \text{'Ski'}\}$$

On remarque que la variable libre a n'est pas utilisée dans la construction du nuplet-résultat (qui ne contient que `l.nom`). On pourrait donc affecter le nuplet `a` à une variable liée, ce qui revient à formuler la requête légèrement différemment : « donnez-moi le nom des logements pour lesquels *il existe une activité* Ski ».

Ce qui donne la formule suivante :

$$\{l.nom | Logement(l) \wedge \exists a(Activit(a) \wedge l.code = a.codeLogement \wedge a.codeActivit = \text{'Ski'})\}$$

On a introduit la sous-formule suivante :

$$\exists a(Activit(a) \wedge l.code = a.codeLogement \wedge a.codeActivit = \text{'Ski'})$$

Cette sous-formule est satisfaite dès que l'on a trouvé *au moins* un nuplet qui satisfait les conditions demandées, à savoir un code activité égal à `Ski`, et le même code logement que celui de la variable l.

Qui dit sous-formule dit logiquement sous-requête en SQL. Voici la syntaxe :

```
select distinct l.nom
from Logement as l
where exists (select ''
              from Activité as a
              where l.code = a.codeLogement
              and a.codeActivité = 'Ski')
```

Le résultat est construit à partir du `select` de premier niveau, qui ne peut accéder qu'à la variable `l`, et pas à la variable (liée) `a`.

Note : La clause du `select` imbriquée ne sert donc absolument à rien d'autre qu'à respecter la syntaxe SQL, et on peut utiliser `select ''`, `select *` ou n'importe quoi d'autre.

Cet exemple montre qu'il est possible d'exprimer une même requête avec des syntaxes différentes, que ce soit au niveau de la formulation en langage naturel ou de l'expression formelle (logique ou SQL).

Les quantificateurs permettent d'imbriquer des formules dans des formules, sans limitation de profondeur. En SQL, on peut de même avoir des imbrications de requêtes sans limitation. La lisibilité et la compréhension en sont quand même affectées.

Prenons une requête un peu plus complexe : je veux les noms des voyageurs qui sont allés dans les Alpes. Une première formulation, complètement « à plat » est la suivante :

```
select distinct v.prénom, v.nom
from Voyageur as v, Séjour as s, Logement as l
where v. idVoyageur=s.idVoyageur
and   s.codeLogement = l .code
and   l.lieu = 'Alpes'
```

Ni la variable `s`, ni la variable `l` ne sont utilisées pour construire le nuplet-résultat. On peut donc l'exprimer ainsi : « je veux les noms des voyageurs pour lesquels il existe un séjour dans les Alpes ». Ce qui donne :

```
select distinct v.prénom, v.nom
from Voyageur as v
where exists (select ''
              from Séjour as s, Logement as l
              where v. idVoyageur=s.idVoyageur
              and   s.codeLogement = l .code
              and   l.lieu = 'Alpes')
```

On pourrait même aller encore plus loin dans l'imbrication avec la requête suivante :

```
select distinct v.prénom, v.nom
from Voyageur as v
where exists (select ''
              from Séjour as s
              where v. idVoyageur=s.idVoyageur
              and exists (select ''
                          from  Logement as l
                           where s.codeLogement = l .code
                           and   l.lieu = 'Alpes')
               )
```

La troisième version correspond à la formulation « Les voyageurs tels *qu'il existe* un de leurs séjours tels que le logement *existe* dans les Alpes ». Elle n'est pas très naturelle, et, de plus, probablement la plus difficile à

comprendre, ce qui ne plaide pas en sa faveur.

3.3.2 Quantificateurs et négation

Il nous reste à découvrir les requêtes probablement les plus complexes, celle où l'on exprime une négation. Voici un premier exemple :on veut les logements qui ne proposent pas de Ski. En reprenant la requête « positive » étudiée précédemment, il suffit d'ajouter une négation devant le quantificateur existentiel.

$$\{l.nom | Logement(l) \wedge \nexists a(Activit(a) \wedge l.code = a.codeLogement \wedge a.codeActivit = \text{'Ski'})\}$$

On a donc formulé la requête en termes logiques : « je veux les logements tels *qu'il n'existe pas* d'activité Ski ». Voici la requête SQL.

```
select distinct l.nom
from Logement as l
where not exists (select ''
                  from Activité as a
                  where l.code = a.codeLogement
                  and a.codeActivité = 'Ski')
```

C'est la seule manière de l'exprimer correctement. Elle donne le résultat suivant :

nom
Causses
U Pinzutu
Tabriz

Vous devriez être convaincus que la requête suivante est très différente (et ne correspond pas à ce que l'on souhaite). L'opérateur != signifie *différent de* en SQL.

```
select l.nom
from Logement as l
where  exists (select ''
                  from Activité as a
                  where l.code = a.codeLogement
                  and a.codeActivité != 'Ski')
```

Dont le résultat est :

nom
Causses
Génépi
U Pinzutu

Réfléchissez au sens de cette requête, trouvez le résultat sur notre petite base. Rappelez-vous que les quantificateurs servent à exprimer une condition sur un ensemble de nuplets, pas sur chaque nuplet en particulier.

Le `not exists` est la porte d'entrée pour exprimer le quantificateur universel. Supposons que l'on cherche les voyageurs qui sont allés dans *tous* les logements. On reformule cette requête avec deux négations : on cherche les voyageurs tels *qu'il n'existe pas* de logement où *ils ne sont pas* allés.

```
select distinct v.prénom, v.nom
from Voyageur as v
where  not exists (select ''
                   from Logement as l
                   where not exists  (select ''
                                      from Séjour as s
                                      where l.code = s.codeLogement
                                      and   v.idVoyageur = s.idVoyageur)
                   )
```

Vous devriez obtenir :

prénom	nom
Nicolas	Bouvier

Vous savez maintenant tout sur la version déclarative de SQL, qui n'est rien d'autre qu'une syntaxe concrète pour exprimer des formules ouvertes sur une base de données. Tout ce qui peut s'exprimer par une formule logique est exprimable en SQL. Ni plus, ni moins. Inversement, tout ce qui ne s'exprime pas par une formule (boucles, incrémentations, etc.) ne s'exprime pas en SQL.

Dans le prochain chapitre, nous verrons la version procédurale, mais il est important de préciser qu'elle n'apporte *rien* en terme de possibilités d'expression. En d'autres termes, vous avez déjà, avec ce que nous venons d'étudier, la capacité d'exprimer toutes les requêtes possibles (à l'exception des agrégations). La version procédurale n'est qu'une manière alternative de concevoir l'interrogation d'une base relationnelle.

Prenez le temps de bien maîtriser ce qui précède, car la compréhension du *sens* de ce que l'on exprime avec les formules de logique des prédicats est la condition nécessaire et suffisante pour utiliser correctement SQL.

3.3.3 Quiz

3.4 S4 : Conception d'une requête SQL

Supports complémentaires :

— Diapositives: SQL: construction d'une requête
— Vidéo sur la construction d'une requête SQL

Vous devriez à ce stade connaître et comprendre l'interprétation d'une requête SQL. Redonnons-la encore une fois sous une forme un peu différente :

— Le *résultat* d'une requête est une relation constituée de nuplets.
— Chaque nuplet du résultat est construit à partir d'un ensemble de n nuplets $t_1, t_2, \cdots, t_n$ *provenant de la base de données*.
— Ces n nuplets doivent satisfaire un ensemble de *conditions* (exprimé par une formule) :.

La *construction* d'une requête consiste

— à indiquer de quels nuplets $t_1, t_2, \cdots, t_n$ nous avons besoin, et d'où chacun provient (c'est la clause `from`)
— à exprimer les conditions avec la clause `where`
— à indiquer comment on construit un nuplet du résultat avec la clause `select`.

C'est tout. Le système pour sa part se charge de trouver toutes les combinaisons possibles des $t_1, t_2, \cdots, t_n$, de tester les conditions, de construire le résultat. Le tout en choisissant la méthode la plus efficace.

Nous sommes maintenant en mesure de tenter de décrire le processus mental qui nous permet de construire une requête SQL pour répondre à un besoin donné. Le processus que nous décrivons s'appuie sur une vision de la structure de la base qui comprend, au minimum, la liste des tables, leurs clés primaires et les clés étrangères. On établit cette vision à partir du schéma, comme le montre par exemple la Fig. 3.1 pour trois tables de la base des films. La bonne connaissance du schéma, et sa compréhension, sont des pré-requis pour exprimer des requêtes SQL correctes.

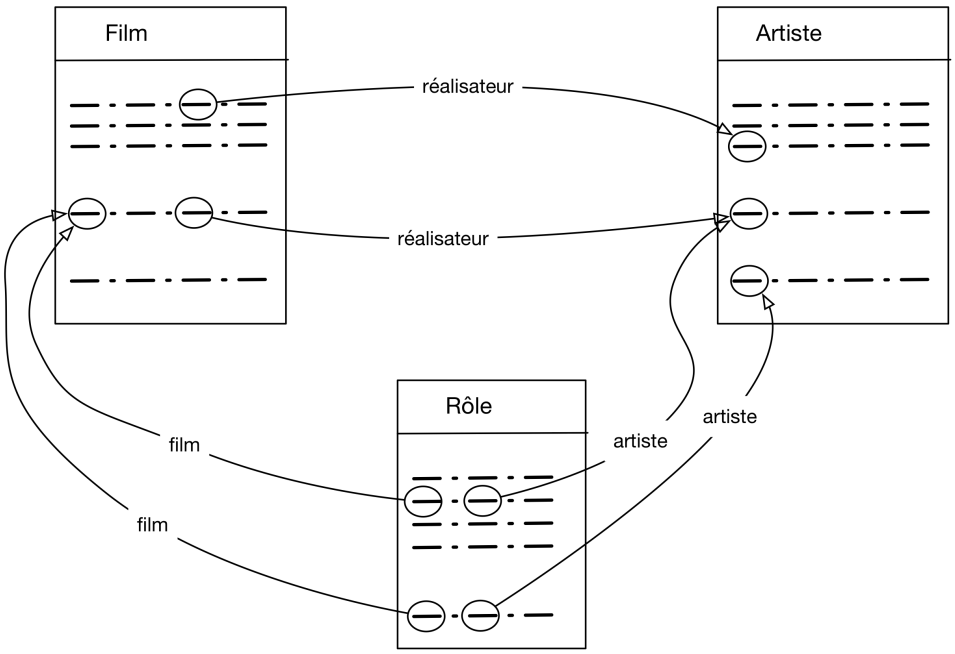

Fig. 3.1 – La base des films « vue » comme un graphe dont les arêtes sont les liens clé étrangère - clé primaire.

Commençons par les requêtes conjonctives, dans lesquelles la principale difficulté est de construire les jointures.

Important : La méthode décrite ci-dessus repose sur la forme déclarative de SQL que nous avons étudiée dans ce chapitre. Le chapitre prochain présentera une approche alternative, basée sur des opérations, qui est à mon avis beaucoup moins adéquate pour apprendre à maîtriser SQL.

3.4.1 Conception d'une jointure

Le mécanisme de base consiste donc à se représenter les nuplets qui permettront de construire un des nuplets du résultat. Dans les cas les plus simples, un seul suffit. Pour la requête « Quelle est l'année de naissance de G. Depardieu » par exemple, on construit un nuplet du résulat à partir d'un nuplet de la table Artiste, dont l'attribut « nom » est « Depardieu », et dont l'attribut « âge » est l'information qui nous intéresse. On désigne ce nuplet par un nom, par exemple *a*. L'image mentale à construire est celle de la Fig. 3.2.

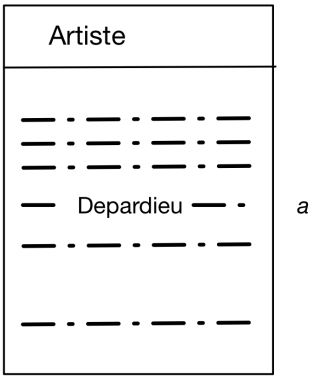

Fig. 3.2 – Interrogation avec un seul nuplet

C'est très élémentaire (pour l'instant) mais toute la requête SQL est déjà codée dans cette représentation.

— Chaque nuplet désigné doit être défini dans le `from`.
— Les contraintes satisfaites par ce nuplet constituent le `where` (nom="Depardieu").
— La clause `select` est toujours triviale (on choisit les attributs à conserver).

Ce qui donne sur ce premier exemple :

```
select anneeNaissance
from Artiste as a
where a.nom='Depardieu'
```

Entrons dans le vif du sujet avec la requête « Titre des films avec pour acteur Depardieu ». Cette fois l'image mentale à construire est celle de la Fig. 3.3. Nous avons besoin, pour construire chaque nuplet du résultat, de trois nuplets de la base : un film, un artiste, un rôle. Dès que nous avons plusieurs nuplets, il faut indiquer de quelle manière ils sont liés : ici les liens sont (comme à peu près toujours) définis par le critère d'égalité des clés primaires et clés étrangères.

On a donné un nom à chaque nuplet, soit *f*, *r* et *a*. La construction de la requête s'ensuit quasiment automatiquement.

```
select f.titre
from Artiste as a, Rôle as r, Film as f
where a.nom='Depardieu'
and a.idArtiste = r.idActeur
and r.idFilm = f.idFilm
```

Notez que les contraintes sur les nuplets sont soit des égalités entre attributs, soit l'égalité entre un attribut

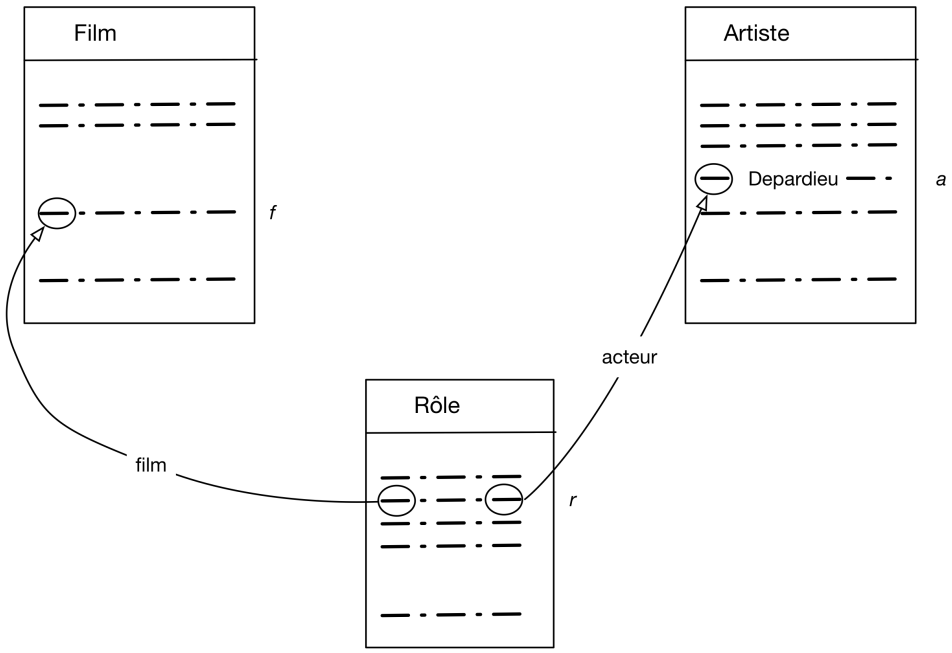

Fig. 3.3 – Les nuplets impliqué dans la recherche des films avec Depardieu

et une constante. Quand nous ajouterons la négation, un troisième type de contrainte apparaîtra, celui de l'existence ou non d'un résultat pour une sous-requête.

Remarquez également comment on se repose sur l'interpéteur SQL pour faire l'essentiel du travail : trouver les nuplets satisfaisant les constraintes, énumérer toutes les combinaisons valides à partir de la base, et construire le résultat.

Voici un exemple un peu plus compliqué qui ne change rien au raisonnement : on veut les titres de film avec Depardieu et Deneuve. L'image à construire est celle de la Fig. 3.4. Ici il faut concevoir qu'il nous faut deux nuplets de la table Artiste, l'un avec pour nom Depardieu (*a1*), et l'autre avec pour nom Deneuve (*a2*). Ces deux nuplets sont liés à deux nuplets *distincts* de la table Rôle, nommons-les *r1* et *r2*. Ces deux derniers nuplets sont liés au *même* film *f* .

À partir de la Fig. 3.4, la construction syntaxique de la requête SQL est encore une fois directe : énumération des variables-nuplets dans le `from`, contraintes dans le `where`, clause `select` selon les besoins.

```
select *
from Artiste as a1, Artiste as a2, Rôle as r1, Rôle as r2, Film as f
where a1.nom='Depardieu'
and a2.nom='Deneuve'
and a1.idArtiste = r1.idActeur
and a2.idArtiste = r2.idActeur
and r1.idFilm = f.idFilm
and r2.idFilm = f.idFilm
```

Voici deux exemples complémentaires. Le premier recherche les films réalisés par Q. Tarantino en 1994. L'image mentale est celle de la Fig. 3.5.

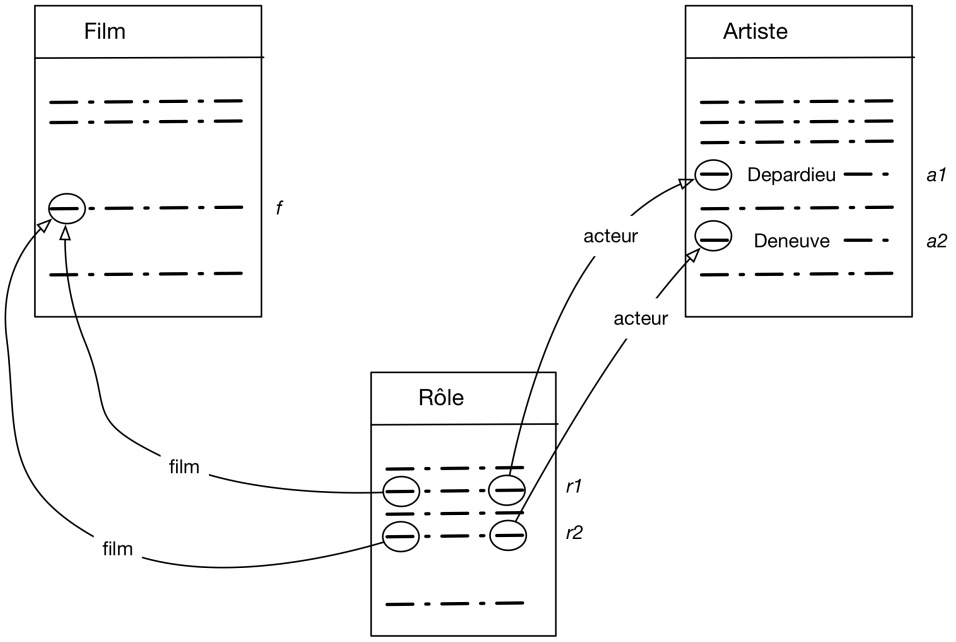

Fig. 3.4 – Les nuplets impliqué dans la recherche des films avec Depardieu et Deneuve

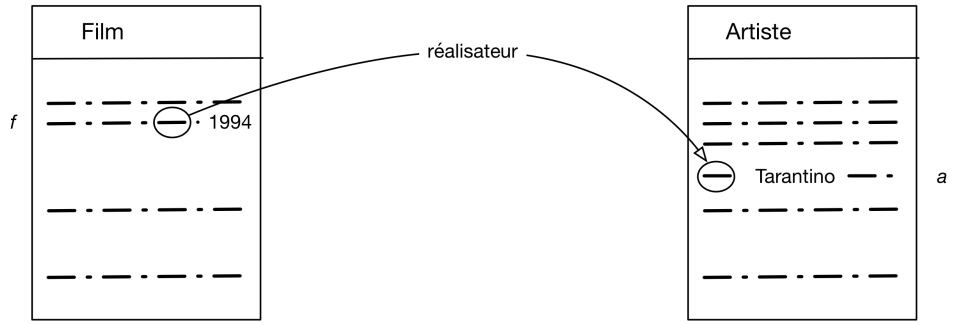

Fig. 3.5 – Recherche les films réalisés par Q. Tarantino en 1994

La requête correspondante est bien entendu celle-ci.

```
select *
from Artiste as a, Film as f
where a.nom='Tarantino'
and  f.année = 1994
and a.idArtiste = f.idRéalisateur
```

Le second exemple recherche les films réalisés par Q. Tarantino en 1994 dans lesquels il joue lui-même dans tant qu'acteur. Je vous laisse étudier et interpréter la Fig. 3.6 et exprimer vous-même la requête SQL.

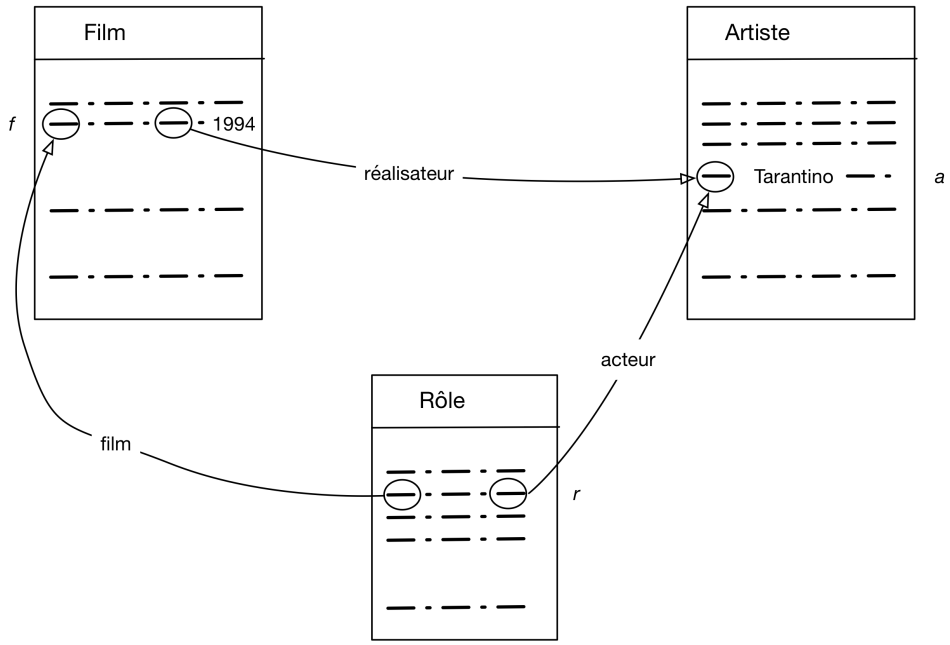

Fig. 3.6 – Recherche les films réalisés par Q. Tarentino en 1994 dans lequels il joue

3.4.2 Conception des requêtes imbriquées

Que se passe-t-il en cas de requête imbriquée, et surtout en cas de nécessité d'exprimer une négation ? Les principes précédents restent valables : on identifie les nuplets de la base qui permettent de produire un nuplet du résultat, on construit la requête comme précédemment, *et la requête imbriquée n'est qu'une contrainte supplémentaire sur ces nuplets*. La seule particularité des requêtes imbriquées est que la contrainte porte sur un ensemble, et pas sur une valeur atomique.

Prenons un exemple : je veux les titres de film avec Catherine Deneuve mais sans Gérard Depardieu. On commence par la solution partielle qui consiste à trouver les films avec Deneuve

```
select f.titre
from Film as f, Rôle as r, Artiste as a
where f.idFilm=r.idFilm
```

(suite sur la page suivante)

(suite de la page précédente)

```
and r.idActeur = a.idArtiste
and a.nom='Deneuve'
```

Maintenant on ajoute la contrainte suivante sur le film f : `dans l'ensemble des acteurs du film *f*, on ne doit pas trouver Gérard Depardieu`. L'ensemble des acteurs du film f qui se nomment Depardieu est obtenu par une requête fonction de f, cette requête est ajoutée dans le `where` et on obtient la requête complète

```
select f.titre
from Film as f, Rôle as r, Artiste as a
where f.idFilm=r.idFilm
and r.idActeur = a.idArtiste
and a.nom='Deneuve'
and not exists (select * from Rôle as r2, Artiste as a2
                where f.idFilm=r2.idFilm and r2.idActeur=a2.idActeur
                and a2.nom='Depardieu')
```

Il faut bien être conscient que cette condition supplémentaire porte sur le film f, et que f doit impérativement intervenir dans la requête imbriquée. La requête suivante par exemple est fausse :

```
select f.titre
from Film as f, Rôle as r, Artiste as a
where f.idFilm=r.idFilm
and r.idActeur = a.idArtiste
and a.nom='Deneuve'
and not exists (select * from Rôle as r2, Artiste as a2
                where r2.idActeur=a2.idActeur
                and a2.nom='Depardieu')
```

La requête imbriquée est ici indépendante des nuplets de la variable principale, et on peut donc évaluer son résultat dès le début : soit il existe un acteur nommé Depardieu (*quel que soit le film*), le `not exists` est toujours faux et le résultat est toujours vide ; soit il n'en existe pas, le `not exists` est toujours vrai et ne sert donc à rien.

3.4.3 La disjonction

Reste à discuter de la disjonction. Il existe une propriété assez utile des formules logiques : on peut toujours les mettre sous une forme dite « normale disjonctive », autrement dit comme la disjonction de conjonctions (voir les exercices). En pratique cela implique que toute requête comprenant un « ou » peut s'écrire comme l'union de requêtes écrites sans « ou ». Cherchons les films avec Deneuve ou Depardieu.

```
select f.titre
from Film as f, Rôle as r, Artiste as a
where f.idFilm=r.idFilm
and r.idActeur = a.idArtiste
and a.nom='Deneuve'
```

(suite sur la page suivante)

(suite de la page précédente)

```
  union
select f.titre
from Film as f, Rôle as r, Artiste as a
where f.idFilm=r.idFilm
and r.idActeur = a.idArtiste
and a.nom='Deneuve'
```

Ce n'est pas très concis. Il est à peu près toujours possible de trouver une formulation plus condensée avec le « or ». Ici ce serait :

```
select f.titre
from Film as f, Rôle as r, Artiste as a
where f.idFilm=r.idFilm
and r.idActeur = a.idArtiste
and (a.nom='Deneuve' or nom='Depardieu')
```

Il n'existe pas de règle générale permettant de trouver la bonne formulation sans réfléchir. La bonne maîtrise des principes de logique, d'équivalence de formule et d'interprétation sont les connaissances clés.

Les principes exposés ici sont très importants. Même s'ils peuvent vous sembler parfois éloignés de vos objectifs pratiques, tout ce qui précède devrait j'espère vous convaincre que maîtriser SQL, c'est d'abord être capable d'aborder la formulation des requêtes de manière rigoureuse, pas de produire une syntaxe finalement relativement simple. À vous de jouer.

3.4.4 Quiz

3.5 Exercices

Exercice Ex-calcul-1 : un peu de réécriture

Il est possible de mettre toute formule en forme normale conjonctive (FNC) $(F_1) \wedge (F_2) \wedge \cdot \wedge F_n$ où les F_i sont des disjonctions de propositions, de la forme $p \vee q \vee ... \vee u$. La négation n'est possible que devant une proposition. Utilisez les règles d'équivalence pour obtenir la FNC des formules suivantes

— $((a \wedge b) \vee (q \wedge r)) \vee z$

Application : on vous demande de trouver les films qui satisfont les critères suivants : soit ils ont été tournés en France, soit ils ont été tournés en Espagne après 2010.

— (pays = “France”) ou (pays=”Espagne” et année > 2010)

Réécrivez cette expression en FNC.

Exercice Ex-calcul-2 : le ou exclusif

Quelle formule propositionnelle exprime le *ou* exclusif entre p et q ?

Exercice Ex-calcul-3 : un peu de raisonnement

Un connecteur peu utilisé en base de données est l'implication, noté $\rightarrow$. L'implication n'existe pas directement en SQL, mais la table des valeurs de vérité de $p \rightarrow q$ est la même que celle de $\neg p \vee q$, ce qui permet de faire du raisonnement en SQL avec un peu de réécriture.

- Donner la table de vérité $p \rightarrow q$. Notez la valeur de vérité de $p \rightarrow q$ quand p et q sont faux et quand p est faux et q est vrai. Est-ce cela correspondait à votre intuition ?
- Montrer que $p \rightarrow q$ est équivalent à $\neg q \rightarrow \neg p$.
  Aide : deux possibilités. La première est de construire la table de vérité (laborieux), la seconde est d'utiliser la premère équivalence, $\neg p \vee q$ (plus élégant).
- Montrer que $p \rightarrow q$ *n'est pas* équivalent à $\neg p \rightarrow \neg q$. Aide : il faut trouver un contre-exemple.
- Application : prenons une implication du langage courant, par exemple « Si je t'aime, prend garde à toi ». Supposons qu'elle est vérifiée (donc, si Carmen aime Don José, alors Don José prend garde à lui). Quelles sont les formes équivalentes ?
  - Si Carmen n'aime pas Don José, alors Don José ne prend pas garde
  - Si Don José ne prend pas garde, c'est que Carmen ne l'aime pas
  - Carmen aime Don José, ou Don José prend garde à lui
  - Carmen n'aime pas Don José, ou Don José prend garde à lui

  Vous pouvez appliquer la question à toute implication de votre choix (si tu me cherches, tu vas me trouver ; si tu vas à Rio, alors n'oublie pas de monter là-haut ; si j'avais un marteau je frapperais le jour, je frapperais la nuit ; si c'est flou, y'a un loup ; etc.)
- Et pour finir sur le sujet, nous savons maintenant exprimer l'implication en SQL. Prenons par exemple l'affirmation « Si le logement est un gîte, il a moins de 5 chambres ». Quelle est la requête qui cherche tous les faits vérifiant cette affirmation ? Et donc comment vérifier si cette affirmation est toujours vraie ?

Exercice Ex-calcul-3bis : aidons les Dupondt à bien raisonner

Supposons les prédicats suivants :

- $\mathrm{Competence(p, c)}$ indique que la personne p a la compétence c
- $\mathrm{OffreEmploi(o, c)}$ indique que l'organisation o offre un emploi pour la compétence c
- $\mathrm{FaitLaffaire(p, o)}$ indique que la personne p fait l'affaire pour un emploi dans l'organisation o.

On suppose que la règle r suivante est toujours vraie :

$$\mathrm{OffreEmploi(o, c)} \wedge \mathrm{Competence(p, c)} \rightarrow \mathrm{FaitLaffaire(p, o)}$$

Voici une petite mise en application des règles de raisonnement que nous avons abordées. Si vous n'arrivez pas à exprimer une démonstration mathématique, essayez au moins de formuler des réponses intuitives (mais claires).

- Reformulez la règle r sans implication
- Dans la figure ci-dessous, les Dupondt peuvent-ils considérer de manière certaine que le capitaine Haddock les traite de clowns ? Formulez la phrase du capitaine selon les prédicats logiques ci-dessus, et vérifiez si on peut en déduire que les Dupondt ont la compétence "clown".

Fig. 3.7 – La première expression du capitaine Haddock, apparemment insultante pour les Dupondt

À tort ou a raison, les Dupondt se sentent offensés. Le capitaine Haddock propose une reformulation (figure ci-dessous).

Fig. 3.8 – Les excuses (?) du capitaine Haddock

Cette reformulation revient-elle vraiment à dire que les Dupondt *ne sont pas* des clowns ?

Quelle serait la bonne formulation d'une phrase qui indiquerait explicitement que le capitaine ne considère pas les Dupondt comme des clowns ?

Exercice Ex-calcul-4 : des équivalences

Voici quelques critères de recherche exprimés sous deux formes en langage naturel. Pour chacune : indiquez si elles sont équivalentes, et donnez la clause `where` correspondante,

Aide : formaliser chaque sous la forme d'une formule propositionnelle, puis regarder si les deux formules sont équivalentes à l'aide des règles d'équivalence.

Exercice Ex-calcul-4bis : encore des équivalences

On dispose de deux prédicats $Auteur(o, x)$ et $Prop(o, x)$ qui sont vrais si, respectivement, x est auteur ou x est propriétaire d'une œuvre d'art o.

— Quelle formule exprime la condition « Soit x est propriétaire, soit x est auteur mais pas les deux » (pour une même œuvre o)
— Quelle est la négation de l'énoncé : « Soit x est propriétaire, soit x est auteur » (idem)
— Comment exprimer l'énoncé suivant sans implication : « Si x est auteur, alors x n'est pas propriétaire » (idem ; utiliser uniquement les connecteurs de SQL : and, or et not).

Exercice Ex-calcul-5 : la quantification universelle

Nous cherchons les logements dans lesquels tous les voyageurs sont allés. Quelle la formulation équivalente sans quantification universelle.

— Les logements tel qu'il existe un voyageur qui y a effectué un séjour
— Les logements tel qu'il n'existe pas de séjour qui n'a pas lieu dans ce logement
— Les logements tel qu'il n'existe pas de voyageur qui n'a pas séjourné dans ce logement
— Les logements tel qu'il n'existe pas de séjour qui n'a pas concerné un voyageur

Donnez la requête SQL correspondant à la bonne formulation

Exercice Ex-calcul-6 : encore des dépendances fonctionnelles

Comment vérifier une DF avec SQL ? Prenons une table `R(A,B)` et la dépendance fonctionnelle $A \to B$

— Quelle est la formule qui exprime cette DF ? (On peut utiliser l'implication maintenant que nous connaissons cet opérateur)
— Donnez la requête SQL qui vérifie que cette DF n'est pas violée (aide : donner la requête qui donne les cas où $A \to B$ n'est pas vérifée : le résultat devrait être vide !).

Exercice Ex-calcul-7 : interprétons SQL

On reprend la requête constituant la liste des logements avec leurs activités, légèrement modifiée.

```
select nom, codeActivité
from   Logement as l, Activité as a, Séjour as s
where  l.code = a.codeLogement
```

Qu'obtient-on dans les trois cas suivants :
- la table `Séjour` contient 1 nuplet,
- la table `Séjour` contient 100 000 nuplets,
- la table `Séjour` est vide.

Exercice Ex-calcul-8 : interprétons SQL, suite

Soit trois tables `R`, `S` et `T` ayant chacune un seul attribut `A`. On veut calculer l'intersection de `R` avec l'union de `S` et `T`, soit $R \cap (S \cup T)$
- La requête suivante est-elle correcte ? Expliquez pourquoi.

```
select r:A
from R as r, S as S, T as t
where r.A=s.A and r.A=t.A
```

- Donnez la bonne requête.
- Faut-il ajouter un `distinct` ?

Exercice Ex-calcul-9 : construisons des requêtes SQL.

Donnes les requêtes SQL correspondant aux Fig. 3.6. `construireSQLQuiz1`, `construireSQLQuiz2` et `construireSQLQuiz3`. À chaque fois, la première clause est `select *` : vous devez compléter avec le `from` et le `where`.

3.6 Atelier : requêtes SQL sur la base Voyageurs

Sur notre base des voyageurs en ligne (ou sur la vôtre, après installation d'un SGBD et chargement de nos scripts), vous devez exprimer les requêtes suivantes :
- Nom des villes
- Nom des logements en Bretagne
- Nom des logements dont la capacité est inférieure à 20
- Description des activités de plongée
- Nom des logements avec piscine
- Nom des voyageurs qui sont allés en Corse
- Les voyageurs qui sont allés ailleurs qu'en Corse
- Nom des logements visités par un auvergnat
- Nom des logements et des voyageurs situés dans la même région
- Les paires de voyageurs (donner les noms) qui ont séjourné dans le même logement
- Les voyageurs qui sont allés (au moins) deux fois dans le même logement
- Les logements qui ont reçu (au moins) deux voyageurs différents
- ...

Contrainte : n'utilisez pas l'imbrication, pour aucune requête (et forcez-vous à utiliser la forme déclarative, même si vous connaissez d'autres options que nous étudierons dans le prochain chapitre).

Pour les requêtes suivantes, en revanche, vous avez droit à l'imbrication (il serait difficile de faire autrement).

— Nom des voyageurs qui ne sont pas allés en Corse
— Noms des voyageurs qui ne vont qu'en Corse s'ils vont quelque part.
— Nom des logements sans piscine
— Nom des voyageurs qui ne sont allés nulle part
— Les logements où personne n'est allé
— Les voyageurs qui n'ont jamais eu l'occasion de faire de la plongée
— Les voyageurs et les logements où ils n'ont jamais séjourné
— Les logements où tout le monde est allé

Vous pouvez finalement reprendre quelques-unes de requêtes précédentes et les exprimer avec l'imbrication.

3.7 Atelier : requêtes sur la base des films

Pour effectuer cet atelier, ouvrez avec un navigateur web le site http://deptfod.cnam.fr/bd/tp/. Aucun droit d'accès n'est nécessaire. Vous accédez à une interface permettant d'entrer des requêtes SQL et de les exécuter sur quelques bases de données.

Pour cet atelier nous vous proposons de travailler sur la base des films. Attention : seules les interrogations sont permises (pas de mise à jour). À droite de la fenêtre dans laquelle vous pouvez entrer les requêtes SQL, vous trouvez le schéma de la base des films. Reportez-vous à ce schéma pour comprendre comment la base est structurée. Juste en dessous, une liste de requêtes vous est proposée : à vous de les exprimer en SQL et de vérifier que votre solution est correcte en l'entrant dans la fenêtre et en l'exécutant.

Si vous ne trouvez pas la solution, ou si vous souhaitez vérifier que votre réponse était la bonne, chaque requête est associée à un bouton « Solution » qui vous permet de voir … la solution. Essayez de résister à la tentation de regarder cette solution trop rapidement.

À vous de jouer !

3.8 Atelier : quand faut-il un `distinct` ?

Difficile… A venir

CHAPITRE 4

SQL, langage algébrique

Le second langage étudié dans ce cours est *l'algèbre relationnelle*. Elle consiste en un ensemble d'opérations qui permettent de manipuler des *relations*, considérées comme des ensembles de nuplets : on peut ainsi faire *l'union* ou la *différence* de deux relations, *sélectionner* une partie des nuplets la relation, effectuer des *produits cartésiens* ou des *projections*, etc.

Important : nous voyons maintenant les relations comme des ensembles, au sens mathématique du terme, avec au moins deux conséquences importantes :

— il n'y pas de doublon dans un ensemble, donc tous les opérateurs éliminent implicitement tout potentiel doublon dansn le résultat ;

— un ensemble n'est pas ordonné : en aucun cas on ne peut donc s'appuyer sur l'hypothèse d'un ordre sur les nuplets.

On peut voir l'algèbre relationnelle comme un langage de programmation très simple qui permet d'exprimer des requêtes sur une base de données relationnelle. C'est donc plus une approche d'informaticien que de logicien. Elle correspond moins naturellement à la manière dont on *pense* une requête. À l'origine, le langage SQL était d'ailleurs entièrement construit sur la logique mathématique, comme nous l'avons vu dans le chapitre *SQL, langage déclaratif*, à l'exception de l'union et de l'intersection. L'algèbre n'était utilisée que comme un moyen de décrire les opérations à effectuer pour évaluer une requête. Petit à petit, les évolutions de la norme SQL ont introduit dans le langage les opérateurs de l'algèbre. Il est maintenant possible de les retrouver tous et d'exprimer toutes les requêtes (plus ou moins facilement) avec cette approche. C'est ce que nous étudions dans ce chapitre.

Note : La base utilisée comme exemple dans ce chapitre est celle de nos intrépides voyageurs, présentée dans le chapitre *Le modèle relationnel*.

4.1 S1 : Les opérateurs de l'algèbre

Supports complémentaires :
— Diapositives: les opérateurs de l'algèbre
— Vidéo sur les opérateurs de l'algèbre

L'algèbre se compose d'un ensemble d'opérateurs, parmi lesquels 6 sont nécessaires et suffisants et permettent de définir les autres par composition. Une propriété fondamentale de chaque opérateur est qu'il prend une ou deux relations en entrée, et produit une relation en sortie. Cette propriété (dite de *clôture*) permet de *composer* des opérateurs : on peut appliquer une sélection au résultat d'un produit cartésien, puis une projection au résultat de la sélection, et ainsi de suite. En fait on peut construire des *expressions algébriques* arbitrairement complexes qui permettent d'effectuer toutes les requêtes relationnelles à l'aide d'un petit nombre d'opérations de base.

Ces opérations sont donc :
— La sélection, dénotée σ
— La projection, dénotée π
— Le renommage, dénoté ρ
— Le produit cartésien, dénoté $\times$
— L'union, $\cup$
— La différence, $-$

Les trois premiers sont des opérateurs *unaires* (ils prennent en entrée une seule relation) et les autres sont des opérateurs *binaires*. À partir de ces opérateurs il est possible d'en définir d'autres, et notamment la *jointure*, $\bowtie$, qui est la composition d'un produit cartésien et d'une sélection. C'est une opération essentielle, nous lui consacrons la prochaine session.

Ces opérateurs sont maintenant présentés tour à tour.

4.1.1 La projection, π

La projection $\pi_{A_1,A_2,\ldots,A_k}(R)$ s'applique à une relation R, et construit une relation contenant tous les nuplets de R, dans lesquels seuls les attributs $A_1, A_2, \ldots A_k$ sont conservés. La requête suivante construit une relation avec le nom des logements et leur lieu.

$$\pi_{nom,lieu}(Logement)$$

On obtient le résultat suivant, après suppression des colonnes `id`, `capacité` et `type` :

nom	lieu
Causses	Cévennes
Génépi	Alpes
U Pinzutu	Corse
Tabriz	Bretagne

En SQL, le projection s'exprime avec le `select` suivi de la liste des attributs à projeter.

```
select nom, lieu
from Logement
```

C'est un habillage syntaxique direct de la projection.

Si on souhaite conserver tous les attributs, on peut éviter d'en énumérer la liste en la remplaçant par *.

```
select *
from Logement
```

Note : En algèbre cette requête est tout simplement l'identité : R

4.1.2 La sélection, σ

La sélection $\sigma_F(R)$ s'applique à une relation, R, et extrait de cette relation les nuplets qui satisfont un critère de sélection, F. Ce critère peut être :

- La comparaison entre un attribut de la relation, A, et une constante a. Cette comparaison s'écrit $A\Theta a$, où Θ appartient à $\{=, <, >, \leq, \geq\}$.
- La comparaison entre deux attributs A_1 et A_2, qui s'écrit $A_1\Theta A_2$ avec les mêmes opérateurs de comparaison que précédemment.

Premier exemple : exprimer la requête qui donne tous les logements en Corse.

$$\sigma_{lieu='Corse'}(Logement)$$

On obtient donc le résultat :

code	nom	capacité	type	lieu
pi	U Pinzutu	10	Gîte	Corse

La sélection a pour effet de supprimer des nuplets, mais chaque nuplet garde l'ensemble de ses attributs. Il ne peut pas y avoir de problème de doublon (pourquoi ?) et il ne faut donc surtout par appliquer un `distinct`.

En SQL, les critères de sélection sont exprimés par la clause `where`.

```
select *
from Logement
where lieu = 'Corse'
```

Les chaînes de caractères doivent impérativement être encadrées par des apostrophes simples, sinon le système ne verrait pas la différence avec un nom d'attribut. Ce n'est pas le cas pour les numériques, car aucun nom d'attribut ne peut commencer par un chiffre.

```
select *
from Logement
where capacité = 134
```

Note : Vous noterez que SQL appelle `select` la projection, et `where` la sélection, ce qui est pour le moins infortuné. Dans des langages modernes comme XQuery (pour les modèles basés sur XML) le, `select` est remplacé par `return`. En ce qui concerne SQL, la question a donné lieu (il y a longtemps) à des débats mais il était déjà trop tard pour changer.

4.1.3 Le produit cartésien, $\times$

Le premier opérateur binaire, et le plus utilisé, est le produit cartésien, $\times$. Le produit cartésien entre deux relations R et S se note $R \times S$, et permet de créer une nouvelle relation où chaque nuplet de R est associé à chaque nuplet de S.

Voici deux relations, la première, R, contient

A	B
a	b
x	y

et la seconde, S, contient :

C	D
c	d
u	v
x	y

Et voici le résultat de $R \times S$:

A	B	C	D
a	b	c	d
a	b	u	v
a	b	x	y
x	y	c	d
x	y	u	v
x	y	x	y

Le nombre de nuplets dans le résultat est exactement $|R| \times |S|$ ($|R|$ dénote le nombre de nuplets dans la relation R).

En lui-même, le produit cartésien ne présente pas un grand intérêt puisqu'il associe aveuglément chaque nuplet de R à chaque nuplet de S. Il ne prend vraiment son sens qu'associé à l'opération de sélection, ce qui permet d'exprimer des *jointures*, opération fondamentale qui sera détaillée plus loin.

En SQL, le produit cartésien est un opérateur `cross join` intégré à la clause `from`.

```
select *
from R cross join S
```

C'est la première fois que nous rencontrons une expression à l'intérieur du `from` en lieu et place de la simple énumération par une virgule. Il y a une logique certaine à ce choix : dans la mesure où `R cross join S` définit une nouvelle relation, la requête SQL peut être vue comme une requête sur cette seule relation, et nous sommes ramenés au cas le plus simple.

Comme illustration de ce principe, voici le résultat du produit cartésien $Logement \times Activit$ (en supprimant l'attribut `description` pour gagner de la place).

code	nom	capacité	type	lieu	codeLogement	codeActivité
ca	Causses	45	Auberge	Cévennes	ca	Randonnée
ge	Génépi	134	Hôtel	Alpes	ca	Randonnée
pi	U Pinzutu	10	Gîte	Corse	ca	Randonnée
ta	Tabriz	34	Hôtel	Bretagne	ca	Randonnée
ca	Causses	45	Auberge	Cévennes	ge	Piscine
ge	Génépi	134	Hôtel	Alpes	ge	Piscine
pi	U Pinzutu	10	Gîte	Corse	ge	Piscine
ta	Tabriz	34	Hôtel	Bretagne	ge	Piscine
ca	Causses	45	Auberge	Cévennes	ge	Ski
ge	Génépi	134	Hôtel	Alpes	ge	Ski
pi	U Pinzutu	10	Gîte	Corse	ge	Ski
ta	Tabriz	34	Hôtel	Bretagne	ge	Ski
ca	Causses	45	Auberge	Cévennes	pi	Plongée
ge	Génépi	134	Hôtel	Alpes	pi	Plongée
pi	U Pinzutu	10	Gîte	Corse	pi	Plongée
ta	Tabriz	34	Hôtel	Bretagne	pi	Plongée
ca	Causses	45	Auberge	Cévennes	pi	Voile
ge	Génépi	134	Hôtel	Alpes	pi	Voile
pi	U Pinzutu	10	Gîte	Corse	pi	Voile
ta	Tabriz	34	Hôtel	Bretagne	pi	Voile

C'est une relation (tout est relation en relationnel) et on peut bien imaginer interroger cette relation comme n'importe quelle autre. C'est exactement ce que fait la requête SQL suivante.

```
select *
from Logement cross join Activité
```

Jusqu'à présent, le `from` ne contenait que des relations « *basées* » (c'es-à-dire stockées dans la base). Maintenant, on a placé une relation *calculée*. Le principe reste le même. Rappelons que l'algèbre est un langage *clos* : il s'applique à des relations et produit une relation en sortie. Il est donc possible d'appliquer à nouveau des opérateurs à cette relation-résultat. C'est ainsi que l'on construit des expressions, comme nous allons le voir dans la session suivante. Nous retrouverons une autre application de cette propriété extrêmement utile quand nous étudierons les vues (chapitre *Schémas relationnel*).

4.1.4 Renommage

Quand les schémas des relations R et S sont complètement distincts, il n'y a pas d'ambiguité sur la provenance des colonnes dans le résultat. Par exemple on sait que les valeurs de la colonne A dans $R \times S$ viennent de la relation R. Il peut arriver (il arrive de fait très souvent) que les deux relations aient des attributs qui ont le même nom. On doit alors se donner les moyens de distinguer l'origine des colonnes dans la relation résultat en donnant un nom distinct à chaque attribut.

Voici par exemple une relation T qui a les mêmes noms d'attributs que R.

A	B
m	n
o	p

Le schéma du résultat du produit cartésien $R \times T$ a pour schéma (A, B, A, B) et présente donc des ambiguités, avec les colonnes A et B en double.

La première solution pour lever l'ambiguité est d'adopter une convention par laquelle chaque attribut est préfixé par le nom de la relation d'où il provient. Le résultat de $R \times T$ devient alors :

R.A	R.B	T.A	T.B
a	b	m	n
a	b	o	p
x	y	m	n
x	y	o	p

Cette convention pose quelques problèmes quand on crée des expressions complexes. Il existe une seconde possibilité, plus générale, pour résoudre les conflits de noms : le *renommage*. Il s'agit d'un opérateur particulier, dénoté ρ, qui permet de renommer un ou plusieurs attributs d'une relation. L'expression $\rho_{A \to C, B \to D}(T)$ permet ainsi de renommer A en C et B en D dans la relation T. Le produit cartésien

$$R \times \rho_{A \to C, B \to D}(T)$$

ne présente alors plus d'ambiguités. Le renommage est une solution très générale, mais asez lourde à utiliser

Il est tout à fait possible de faire le produit cartésien d'une relation avec elle-même. Dans ce cas le renommage où l'utilisation d'un préfixe distinctif est impératif. Voici par exemple le résultat de $R \times R$, dans lequel on préfixe par $R1$ et $R2$ respectivement les attributs venant de chacune des opérandes.

R1.A	R1.B	R1.A	R2.B
a	b	a	b
a	b	x	y
x	y	a	b
x	y	x	y

En SQL, le renommage est obtenu avec le mot-clé `as`. Il peut s'appliquer soit à la relation, soit aux attributs (ou bien même aux deux). Le résultat suivant est donc obtenu avec la requête :

```
select *
from R as R1 cross join R as R2
```

On obtient une relation de schéma `(R1.A, R1.B, R1.A, R2.B)`, avec des noms d'attribut qui ne sont en principe pas acceptés par la norme SQL. Il reste à spécifier ces noms en ajoutant dans `as` dans la clause de projection.

```
select R1.a as premier_a, R1.b as premier_b, R2.a as second_a, R2.b as second_b
from R as R1 cross R as R2
```

Ce qui donnera donc le résultat :

premier_a	premier_b	second_a	second_b
a	b	a	b
a	b	x	y
x	y	a	b
x	y	x	y

Sur notre schéma, le renommage s'impose par exemple si on effectue le produit cartésien entre `Voyageur` et `Séjour` car l'attribut `idVoyageur` apparaît dans les deux tables. Essayez la requête :

```
select Voyageur.idVoyageur, Séjour.idVoyageur
from Voyageur cross join Séjour
```

Elle vous renverra une erreur comme *Encountered duplicate field name : "idVoyageur"*. Il faut nommer explicitement les attributs pour lever l'ambiguité.

```
select Voyageur.idVoyageur as idV1, Séjour.idVoyageur as idV2
from Voyageur cross join Séjour
```

4.1.5 L'union, $\cup$

Il existe deux autres opérateurs binaires, qui sont à la fois plus simples et moins fréquemment utilisés.

Le premier est l'union. L'expression $R \cup S$ crée une relation comprenant tous les nuplets existant dans l'une ou l'autre des relations R et S. Il existe une condition impérative : *les deux relations doivent avoir le même schéma*, c'est-à-dire même nombre d'attributs, mêmes noms et mêmes types.

L'union des relations $R(A, B)$ et $S(C, D)$ données en exemple ci-dessus est donc interdite (on ne saurait pas comment nommer les attributs dans le résultat). En revanche, en posant $S' = \rho_{C \to A, D \to B}(S)$, il devient possible de calculer $R \cup S'$, avec le résultat suivant :

A	B
a	b
x	y
c	d
u	v

Comme pour la projection, il faut penser à éviter les doublons. Donc le nuplet `(x,y)` qui existe à la fois dans R et dans S' ne figure qu'une seule fois dans le résultat.

L'union est un des opérateurs qui existe dans SQL depuis l'origine. La requête suivante effectue l'union des lieux de la table `Logement` et des régions de la table `Voyageur`. Pour unifier les schémas, on a projeté sur cet unique attribut, et on a effectué un renommage.

```
select lieu from Logement
  union
select région as lieu from Voyageur
```

On obtient le résultat suivant.

lieu
Cévennes
Alpes
Corse
Bretagne
Auvergne
Tibet

Notez que certains noms comme « Corse » apparaîssent deux fois : vous savez maintenant comment éliminer les doublons avec SQL.

4.1.6 La différence, $-$

Comme l'union, la différence s'applique à deux relations qui ont le même schéma. L'expression $R - S$ a alors pour résultat tous les nuplets de R qui ne sont pas dans S.

Voici la différence de R et S', les deux relations étant définies comme précédemment.

A	B
a	b

En SQL, la différence est obtenue avec `except`.

```
select A, B from R
   except
select C as A, D as B from S
```

La différence est le seul opérateur algébrique qui permet d'exprimer des requêtes comportant une négation (on veut « *rejeter* » quelque chose, on « *ne veut pas* » des nuplets ayant telle propriété). La contrainte d'identité des schémas rend cet opérateur très peu pratique à utiliser, et on lui préfère le plus souvent la construction logique du SQL « déclaratif », `not exists`.

Note : L'opérateur `except` n'est même pas proposé par certains systèmes comme MYSQL.

4.1.7 Quiz

4.2 S2 : la jointure

Supports complémentaires :

— Diapositives: la jointure algébrique
— Vidéo sur la jointure algébrique

Toutes les requêtes exprimables avec l'algèbre relationnelle peuvent se construire avec les 6 opérateurs présentés ci-dessus. En principe, on pourrait donc s'en contenter. En pratique, il existe d'autres opérations, très couramment utilisées, qui peuvent se contruire par composition des opérations de base. La plus importante est la jointure.

4.2.1 L'opérateur ⋈

Afin de comprendre l'intérêt de cet opérateur, regardons le produit cartésien $\mathrm{Logement} \times \mathrm{Activit}$, dont le résultat est rappelé ci-dessous.

code	nom	capacité	type	lieu	codeLogement	codeActivité
ca	Causses	45	Auberge	Cévennes	ca	Randonnée
ge	Génépi	134	Hôtel	Alpes	ca	Randonnée
pi	U Pinzutu	10	Gîte	Corse	ca	Randonnée
ta	Tabriz	34	Hôtel	Bretagne	ca	Randonnée
ca	Causses	45	Auberge	Cévennes	ge	Piscine
ge	Génépi	134	Hôtel	Alpes	ge	Piscine
pi	U Pinzutu	10	Gîte	Corse	ge	Piscine
ta	Tabriz	34	Hôtel	Bretagne	ge	Piscine
ca	Causses	45	Auberge	Cévennes	ge	Ski
ge	Génépi	134	Hôtel	Alpes	ge	Ski
pi	U Pinzutu	10	Gîte	Corse	ge	Ski
ta	Tabriz	34	Hôtel	Bretagne	ge	Ski
ca	Causses	45	Auberge	Cévennes	pi	Plongée
ge	Génépi	134	Hôtel	Alpes	pi	Plongée
pi	U Pinzutu	10	Gîte	Corse	pi	Plongée
ta	Tabriz	34	Hôtel	Bretagne	pi	Plongée
ca	Causses	45	Auberge	Cévennes	pi	Voile
ge	Génépi	134	Hôtel	Alpes	pi	Voile
pi	U Pinzutu	10	Gîte	Corse	pi	Voile
ta	Tabriz	34	Hôtel	Bretagne	pi	Voile

Si vous regardez attentivement cette relation, vous noterez que le résultat comprend manifestement un grand nombre de nuplets qui ne nous intéressent pas. C'est le cas de toutes les lignes pour lesquelles le `code`

(provenant de la table `Logement`) et le `codeLogement` (provenant de la table `Activité`) sont distincts. Cela ne présente pas beaucoup de sens (à priori) de rapprocher des informations sur l'hôtel Génépi, dans les Alpes, avec l'activité de plongée en Corse.

Note : Il est bien sûr arbitraire de dire qu'un résultat « n'a pas de sens » ou « ne présente aucun intérêt ». Nous nous plaçons ici dans un contexte où l'on cherche à reconstruire une information sur certaines entités du monde réel, dont la description a été distribuée dans plusieurs tables par la normalisation. C'est l'utilisation sans doute la plus courante de SQL.

Si, en revanche, on considère le produit cartésien comme un *résultat intermédiaire*, on voit qu'il permet d'associer des nuplets initialement répartis dans des tables distinctes. Sur notre exemple, on rapproche les informations générales sur un logement et la liste des activités de ce logement.

La sélection qui effectue une rapprochement pertinent est celle qui ne conserve que les nuplets partageant la même valeur pour les attributs `code` et `codeLogement`, soit :

$$\sigma_{code=codeLogement}(\text{Logement} \times \text{Activit})$$

Prenez bien le temps de méditer cette opération de sélection : nous ne voulons conserver que les nuplets de $\text{Logement} \times \text{Activit}$ pour lesquelles l'identifiant du logement (provenant de `Logement`) est identique à celui provenant de `Activité`. En regardant le produit cartésien ci-dessous, vous devriez pouvoir vous convaincre que cela revient à conserver les nuplets qui ont un sens : chacune contient des informations sur un logement et sur une activité dans ce *même* logement.

On obtient le résultat ci-dessous.

code	nom	capacité	type	lieu	codeLogement	codeActivité
ca	Causses	45	Auberge	Cévennes	ca	Randonnée
ge	Génépi	134	Hôtel	Alpes	ge	Piscine
ge	Génépi	134	Hôtel	Alpes	ge	Ski
pi	U Pinzutu	10	Gîte	Corse	pi	Plongée
pi	U Pinzutu	10	Gîte	Corse	pi	Voile

On a donc effectué une *composition* de deux opérations (un produit cartésien, une sélection) afin de rapprocher des informations réparties dans plusieurs relations, mais ayant des liens entre elles (toutes les informations dans un nuplet du résultat sont relatives à un seul logement). Cette opération est une *jointure*, que l'on peut directement, et simplement, noter :

$$\text{Logement} \bowtie_{\text{code=codeLogement}} \text{Activit}$$

La jointure consiste donc à rapprocher les nuplets de deux relations pour lesquelles les valeurs d'un (ou plusieurs) attributs sont identiques. De fait, dans la plupart des cas, ces attributs communs sont (1) la clé primaire de l'une des relations et (2) la clé étrangère dans l'autre relation. Dans l'exemple ci-dessus, c'est le cas pour `code` (clé primaire de *Logement*) et `codeLogement` (clé étrangère dans *Activité*).

Note : Le logement Tabriz, qui ne propose pas d'activité, n'apparaît pas dans le résultat de la jointure. C'est normal et conforme à la définition que nous avons donnée, mais peut parfois apparaître comme une

contrainte. Nous verrons dans le chapitre final sur SQL que ce dernier propose une variante, la *jointure externe*, qui permet de la contourner.

La notation de la jointure, $R \bowtie_F S$, est un racourci pour $\sigma_F(R \times S)$.

Note : Le critère de rapprochement, F, peut être n'importe quelle opération de comparaison liant un attribut de R à un attribut de S. En pratique, on emploie peu les $\neq$ ou "<" qui sont difficiles à interpréter, et on effectue des égalités.

Si on n'exprime pas de critère de rapprochement, la jointure est équivalente à un produit cartésien.

Initialement, SQL ne proposait pour effectuer la jointure que la version déclarative.

```
select *
from Logement as l, Activité as a
where l.code=a.codeLogement
```

En 1992, la révision de la norme a introduit l'opérateur algébrique qui, comme le produit cartésien, et pour les mêmes raisons, prend place dans le `from`.

```
select *
from Logement join Activité on (code=codeLogement)
```

Il s'agit donc d'une manière alternative *d'exprimer* une jointure. Laquelle est la meilleure ? Aucune, puisque toutes les deux ne sont que des spécifications, et n'imposent en aucun cas au système une méthode particulière d'exécution. Il est d'ailleurs exclu pour un système d'appliquer aveuglément la définition de la jointure et d'effectuer un produit cartésien, puis une sélection, car il existe des algorithmes d'évaluation bien plus efficaces.

4.2.2 Résolution des ambiguités

Il faut être attentif aux ambiguités dans le nommage des attributs qui peut survenir dans la jointure au même titre que dans le produit cartésien. Les solutions à employer sont les mêmes : on préfixe par le nom de la relation ou par un synonyme, ou bien on renomme des attributs avant d'effectuer la jointure.

Supposons que l'on veuille obtenir les voyageurs et les séjours qu'ils ont effectués. La jointure s'exprime en principe comme suit :

```
select *
from Voyageur join Séjour on (idVoyageur=idVoyageur)
```

Le système renvoie une erreur : La clause de jointure `on (idVoyageur=idVoyageur)` est clairement ambigüe. Pour MySQL, le message est par exemple *Column "idVoyageur" in on clause is ambiguous*. Nouvelle tentative :

```
select *
from Voyageur join Séjour on (Voyageur.idVoyageur=Séjour.idVoyageur)
```

Nouveau message d'erreur (cette fois, sous MySQL : *Encountered duplicate field name : "idVoyageur"*). La liste des noms d'attribut dans le nuplet-résultat obtenu avec `select *` comprend encore deux fois `idVoyageur`.

Première solution : on renomme les attributs du nuplet résultat. Cela suppose d'énumérer tous les attributs.

```
select V.idVoyageur as idV1, V.nom, S.idVoyageur as idV2, début, fin
from Voyageur as V join Séjour as S  on (V.idVoyageur=S.idVoyageur)
```

Cette première solution consiste à effectuer un renommage *après* la jointure. Une autre solution est d'effectuer le renommage *avant* la jointure.

```
select *
from (select idVoyageur as idV1, nom from Voyageur) as V
               join
     (select idVoyageur as idV2, début, fin from Séjour) as S
           on (V.idV1=S.idV2)
```

En algèbre, la requête ci-dessus correspond à l'expression suivante :

$(\rho_{idVoyageur \to idV1}(\pi_{idVoyageur,nom} Voyageur) \bowtie_{idV1=idV2} \rho_{idVoyageur \to idV2}(\pi_{idVoyageur,dbut,fin} Sjour))$

On voit que le `from` commence à contenir des expressions de plus en plus complexes. Dans ses premières versions, SQL ne permettait pas des constructions algébriques dans le `from`, ce qui avait l'avantage d'éviter des constructions qui ressemblent de plus en plus à de la programmation. Rappelons qu'il existe une syntaxe alternative à la requête ci-dessus, dans la forme déclarative de SQL étudiée au chapitre précédent.

```
select V.idVoyageur as idV1, V.nom, S.idVoyageur as idV2, début, fin
from Voyageur as V, Séjour as S
where V.idVoyageur= S.idVoyageur
```

Bref, vous commencez à avoir l'embarras du choix.

La jointure dite « naturelle »

Il reste à vrai dire, avec SQL, un troisième choix, la jointure dite « naturelle ». Elle s'applique uniquement quand les attributs de jointure ont des noms identiques dans les deux tables. C'est le cas ici, (l'attribut de jointure est `idVoyageur`, que ce soit dans `Logement` ou dans `Séjour`). La jointure naturelle s'effectue alors automatiquement sur ces attributs communs, et ne conserve que l'un des attributs dans le résultat, ce qui élimine l'ambiguité. La syntaxe devient alors très simple.

```
select *
from Voyageur as V natural join Séjour
```

Si les attributs de jointures sont nommés différemment, la jointure naturelle devient plus délicate à utiliser puisqu'il faut au préalable effectuer des renommages pour faire coïncider les noms des attributs à comparer.

À partir de là, vous savez comment effectuer plusieurs jointures. Un exemple devrait suffire : supposons que l'on veuille les noms des voyageurs et les noms des logements qu'ils ont visités. La requête algébrique devient un peu compliquée. On va s'autoriser une construction en plusieurs étapes.

Tout d'abord on effectue un renommage sur la table `Voyageur` pour éviter les futures ambiguités.

$$V2 := \rho_{idVoyageur \to idV, nom \to nomVoyageur}(Voyageur)$$

Opération semblable sur les logements.

$$L2 := \rho_{nom \to nomLogement}(Logement)$$

Et finalement, voici la requête algébrique complète, utilisant `V2` et `L2`.

$$\pi_{nomVoyageur, nomLogement}(\text{L2} \bowtie_{\text{code=codeLogement}} \text{Sjour}) \bowtie_{\text{idVoyageur=idV}} \text{V2})$$

En SQL, il faut tout écrire avec une seule requête. Allons-y

```
select nomVoyageur, nomLogement
from ( (select idVoyageur as idV, nom as nomVoyageur from Voyageur) as V
                 join
            Séjour as S on idV=idVoyageur)
             join
         (select code, nom as nomLogement from Logement) as L
             on codeLogement = code
```

Ce n'est pas très lisible… Pour comparaison, la version déclarative de ces jointures.

```
select V.nom as nomVoyageur, L.nom as nomLogement
from   Voyageur as V, Séjour as S, Logement as L
where  V.idVoyageur = S.idVoyageur
and    S.codelogement = L. code
```

À vous de voir quel style (ou mélange des styles) vous souhaitez adopter.

4.2.3 Quiz

4.3 S3 : Expressions algébriques

Supports complémentaires :

- — Diapositives: expressions algébriques
- — Vidéo sur les expressions algébriques

Cette section est consacrée à l'expression de requêtes algébriques complexes impliquant plusieurs opérateurs. On utilise la *composition* des opérations, rendue possible par le fait que tout opérateur produit en sortie une relation sur laquelle on peut appliquer à nouveau des opérateurs.

Note : Les expressions sont seulement données dans la forme concise de l'algèbre. La syntaxe SQL équivalente est à faire à titre d'exercices (et à tester sur notre site).

4.3.1 Sélection généralisée

Regardons d'abord comment on peut généraliser les critères de sélection de l'opérateur σ. Jusqu'à présent on a vu comment sélectionner des nuplets satisfaisant *un* critère de sélection, par exemple : « les logements de type "Hôtel" ». Maintenant supposons que l'on veuille retrouver les hôtels dont la capacité est supérieure à 100. On peut exprimer cette requête par une composition :

$$\sigma_{capacit>100}(\sigma_{type='Htel'}(Logement))$$

Ce qui revient à pouvoir exprimer une sélection avec une *conjonction* de critères. La requête précédente est donc équivalente à celle ci-dessous, où le $\wedge$ dénote le "et".

$$\sigma_{capacit>100 \wedge type='Htel'}(Logement)$$

La composition de plusieurs sélections revient à exprimer une conjonction de critères de recherche. De même la composition de la sélection et de l'union permet d'exprimer la *disjonction*. Voici la requête qui recherche les logements qui sont en Corse, *ou* dont la capacité est supérieure à 100.

$$\sigma_{capacit>100}(Logement) \cup \sigma_{lieu='Corse'}(Logement)$$

Ce qui permet de s'autoriser la syntaxe suivante, où le "$\vee$" dénote le "ou".

$$\sigma_{capacit>100 \ \vee \ lieu='Corse'}(Logement)$$

Enfin la *différence* permet d'exprimer la *négation* et « d'éliminer » des nuplets. Par exemple, voici la requête qui sélectionne les logements dont la capacité est supérieure à 200 mais qui ne sont *pas* aux Antilles.

$$\sigma_{capacit>100}(Logement) - \sigma_{lieu='Corse'}(Logement)$$

Cette requête est équivalente à une sélection où on s'autorise l'opérateur "$\neq$" :

$$\sigma_{capacit>100 \wedge lieu \neq 'Corse'}(Logement)$$

Important : Attention avec les requêtes comprenant une négation, dont l'interprétation est parfois subtile. D'une manière générale, l'utilisation du "$\neq$" *n'est pas* équivalente à l'utilisation de la différence, l'exemple précédent étant une exception. Voir la prochaine section.

En résumé, les opérateurs d'union et de différence permettent de définir une sélection σ_F où le critère F est une expression booléenne quelconque. Attention cependant : si toute sélection avec un "ou" peut s'exprimer par une union, l'inverse n'est pas vrai (exercice).

4.3.2 Requêtes conjonctives

Les requêtes dites *conjonctives* constituent l'essentiel des requêtes courantes. Intuitivement, il s'agit de toutes les recherches qui s'expriment avec des "et", par opposition à celles qui impliquent des "ou" ou des "not". Dans l'algèbre, ces requêtes sont toutes celles qui peuvent s'écrire avec seulement trois opérateurs : π, σ, $\times$ (et donc, indirectement, $\bowtie$).

Les plus simples sont celles où on n'utilise que π et σ. En voici quelques exemples.

— Nom des logements en Corse :
$\pi_{nom}(\sigma_{lieu='Corse'}(Logement))$
— Code des logements où l'on pratique la voile.
$\pi_{codeLogement}(\sigma_{codeActivit='Voile'}(Activit))$
— Nom et prénom des clients corses
$\pi_{nom,prnom}(\sigma_{rgion='Corse'}(Voyageur))$

Des requêtes légèrement plus complexes - et extrêmement utiles - sont celles qui impliquent la jointure. On doit utiliser la jointure dès que les attributs nécessaires pour évaluer une requête sont réparties dans au moins deux relations. Ces « attributs nécessaires » peuvent être :

— Soit des attributs qui figurent dans le résultat ;
— Soit des attributs sur lesquels on exprime un critère de sélection.

Considérons par exemple la requête suivante : « Donner le nom et le lieu des logements où l'on pratique la voile ». Une analyse très simple suffit pour constater que l'on a besoin des attributs `lieu` et `nom` qui apparaîssent dans la relation `Logement`, et de `codeActivité` qui apparaît dans `Activité`.

Donc il faut faire une jointure, de manière à rapprocher les nuplets de `Logement` et de `Activité`. Il reste donc à déterminer le (ou les) attribut(s) sur lesquels se fait ce rapprochement. Ici, comme dans la plupart des cas, la jointure permet de « recalculer » l'association entre les relations `Logement` et *Activité*. Elle s'effectue donc par appariement de la clé primaire d'une part (dans `Logement`), de la clé étrangère d'autre part.

$$\pi_{nom,lieu}(Logement \bowtie_{code=codeLogement} (\sigma_{codeActivit='Voile'}(\text{Activité})))$$

En pratique, la grande majorité des opérations de jointure s'effectue sur des attributs qui sont clé primaire dans une relation, et clé étrangère dans l'autre. Il ne s'agit pas d'une règle absolue, mais elle résulte du fait que la jointure permet le plus souvent de reconstituer le lien entre des informations qui sont naturellement associées (comme un logement et ses activités, ou un logement et ses clients), mais qui ont été réparties dans plusieurs relations au moment de la conception de la base. Voir le chapitre *Conception d'une base de données* à ce sujet.

Voici quelques autres exemples qui illustrent cet état de fait :

— Nom des clients qui sont allés à Tabriz (en supposant connu le code, `ta`, de cet hôtel) :

$$\pi_{nom}(\text{Voyageur} \bowtie_{idVoyageur=idVoyageur} \sigma_{codeLogement='ta'}(\text{Séjour}))$$

— Quels lieux a visité le client 30 :

$$\pi_{lieu}(\sigma_{idVoyageur=30}(\text{Séjour}) \bowtie_{codeLogement=code} (\text{Logement}))$$

— Nom des clients qui ont eu l'occasion de faire de la voile :

$$\pi_{nom}(\texttt{Voyageur} \bowtie_{idVoyageur=idVoyageur} (\texttt{Séjour} \bowtie_{codeLogement=codeLogement} \sigma_{codeActivit='Voile'}(\texttt{Activité})))$$

Note : Pour simplifier un peu l'expression, on a considéré ci-dessus que l'ambiguité sur l'attribut de jointure `idVoyageur` était effacée par la projection finale sur `nom`. En toute rigueur, la relation obtenue par

$$\texttt{Voyageur} \bowtie_{idVoyageur=idVoyageur} (\texttt{Séjour} \bowtie_{codeLogement=codeLogement} \sigma_{codeActivit='Voile'}(\texttt{Activité}))$$

comporte des noms d'attributs doublés auxquels il faudrait appliquer un renommage.

La dernière requête comprend deux jointures, portant à chaque fois sur des clés primaires et/ou étrangères. Encore une fois ce sont les clés qui définissent les liens entre les relations, et elle servent donc naturellement de support à l'expression des requêtes.

Voici maintenant un exemple qui montre que cette règle n'est pas systématique. On veut exprimer la requête qui recherche les noms des clients qui sont partis en vacances dans leur lieu de résidence, ainsi que le nom de ce lieu.

Ici on a besoin des informations réparties dans les relations *Logement*, *Séjour* et *Voyageur*. Voici l'expression algébrique :

$$\pi_{nom,lieu}(\text{Voyageur} \bowtie_{idVoyageur=idVoyageur \wedge rgion=lieu} (\text{Séjour} \bowtie_{codeLogement=code} \text{Logement}))$$

Les jointures avec la relation `Séjour` se font sur les couples (clé primaire, clé étrangère), mais on a en plus un critère de rapprochement relatif à l'attribut `lieu` de `Voyageur` et de `Logement`.

4.3.3 Requêtes avec $\cup$ et $-$

Pour finir, voici quelques exemples de requêtes impliquant les deux opérateurs $\cup$ et $-$. Leur utilisation est moins fréquente, mais elle peut s'avérer absolument nécessaire puisque ni l'un ni l'autre ne peuvent s'exprimer à l'aide des trois opérateurs « conjonctifs » étudiés précédemment. En particulier, la différence permet d'exprimer toutes les requêtes où figure une négation : on veut sélectionner des données qui *ne* satisfont *pas* telle propriété, ou tous les « untels » *sauf* les "x" et les "y", etc.

Illustration concrète sur la base de données avec la requête suivante : quels sont les codes des logements qui *ne* proposent *pas* de voile ?

$$\pi_{code}(\text{Logement}) - \pi_{codeLogement}(\sigma_{codeActivit='Voile'}(\text{Activité}))$$

Comme le suggère cet exemple, la démarche générale pour construire une requête du type « Tous les O qui ne satisfont pas la propriété p » est la suivante :

— Construire une première requête A qui sélectionne tous les O.
— Construire une deuxième requête B qui sélectionne tous les O *qui satisfont* p.
— Finalement, faire $A - B$.

Les requêtes A et B peuvent bien entendu être arbitrairement complexes et mettre en œuvre des jointures, des sélections, etc. La seule contrainte est que le résultat de A et de B comprenne le même nombre d'attributs (et, en théorie, les mêmes noms, mais on peut s'affranchir de cette contrainte).

Important : Attention à ne pas considérer que l'utilisation du comparateur $\neq$ est équivalent à la différence. La requête suivante par exemple *ne donne pas* les logements qui ne proposent pas de voile

$$\pi_{codeLogement}(\sigma_{codeActivit \neq 'Voile'}(\text{Activité}))$$

Pas convaincu(e) ? Réfléchissez un peu plus, faites le calcul concret. C'est l'un de pièges à éviter.

Voici quelques exemples complémentaires qui illustrent ce principe.

— Régions où il y a des clients, mais pas de logement.

$$\pi_{rgion}(\text{Voyageur}) - \pi_{rgion}(\rho_{lieu \to rgion}(\text{Logement}))$$

— Identifiant des logements qui n'ont pas reçu de client tibétain.

$$\pi_{code}(\text{Logement}) - \pi_{codeLogement}(\text{Séjour} \bowtie_{idVoyageur=idVoyageur} \sigma_{rgion='Tibet'}(\text{Voyageur}))$$

— Id des clients qui ne sont pas allés en Corse.

$$\pi_{idVoyageur}(\text{Voyageur}) - \pi_{idVoyageur}(\sigma_{lieu='Corse'}(\text{Logement}) \bowtie_{code=codeLogement} \text{Séjour})$$

La dernière requête construit l'ensemble des `idVoyageur` pour les clients qui ne sont pas allés en Corse. Pour obtenir le nom de ces clients, il suffit d'ajouter une jointure (exercice).

4.3.4 Complément d'un ensemble

La différence peut être employée pour calculer le *complément* d'un ensemble. Prenons l'exemple suivant : on veut les ids des clients *et* les logements où ils ne sont pas allés. En d'autres termes, parmi toutes les associations Voyageur/Logement possibles, on veut justement celles qui *ne sont pas* représentées dans la base !

C'est un des rares cas où le produit cartésien seul est utile : il permet justement de constituer « toutes les associations possibles ». Il reste ensuite à en soustraire celles qui sont dans la base avec l'opérateur $-$.

$$(\pi_{idVoyageur}(\text{Voyageur}) \times \pi_{code}(\text{Logement})) - \pi_{idVoyageur,codeLogement}(\text{Séjour})$$

4.3.5 Quantification universelle

Enfin la différence est nécessaire pour les requêtes qui font appel à la quantification universelle : celles où l'on demande par exemple qu'une propriété soit *toujours* vraie. À priori, on ne voit pas pourquoi la différence peut être utile dans de tels cas. Cela résulte simplement de l'équivalence suivante : une propriété est vraie pour *tous* les éléments d'un ensemble si et seulement si *il n'existe pas* un élément de cet ensemble pour lequel la propriété est *fausse*. La quantification universelle s'exprime par une double négation.

En pratique, on se ramène toujours à la seconde forme pour exprimer des requêtes. Prenons un exemple : quels sont les clients dont *tous* les séjours ont eu lieu en Corse ? On l'exprime également par "quels sont clients pour lesquels *il n'existe pas* de séjour dans un lieu qui soit différent de la Corse. Ce qui donne l'expression suivante :

$$\pi_{idVoyageur}(\text{Séjour}) - \pi_{idVoyageur}(\sigma_{lieu\neq'Corse'}(\text{Séjour}))$$

Pour finir, voici une des requêtes les plus complexes, la *division*. L'énoncé (en français) est simple, mais l'expression algébrique ne l'est pas du tout. L'exemple est le suivant : on veut les ids des clients qui sont allés dans *tous* les logements.

Traduit avec (double) négation, cela donne : les ids des clients tels *qu'il n'existe pas* de logement où ils *ne soient pas* allés. Ce qui donne l'expression algébrique suivante :

$$\pi_{idVoyageur}(\text{Voyageur}) - \pi_{idVoyageur}((\pi_{idVoyageur}(\text{Voyageur}) \times \pi_{code}(\text{Logement})) - \pi_{idVoyageur,idLogement}(\text{Séjour}))$$

Explication : on réutilise l'expression donnant les clients et les logements où ils ne sont pas allés (voir plus haut) :

$$\pi_{idVoyageur}(\text{Voyageur}) \times \pi_{code}(\text{Logement})) - \pi_{idVoyageur,idLogement}(\text{Séjour})$$

On obtient un ensemble B. Il reste à prendre tous les clients, sauf ceux qui sont dans B.

$$\pi_{idVoyageur}(\text{Voyageur}) - B$$

Ce type de requête est rare (heureusement) mais illustre la capacité de l'algèbre à exprimer par de simples manipulations ensemblistes des opérations complexes.

4.3.6 Quiz

4.4 S4 : Complément : évaluation et optimisation

Ce complément introduit la manière dont un SGBD analyse, optimise et exécute une requête. Il est présenté dans le but de vous donner un aperçu de l'utilité de l'algèbre dans un contexte d'exécution de requêtes, mais ne fait pas partie du contenu du cours soumis à examen.

Supports complémentaires :

— Diapositives: introduction à l'optimisation
— Vidéo d'introduction à l'optimisation

SQL étant un langage *déclaratif* dans lequel on n'indique ni les algorithmes à appliquer, ni les chemins d'accès aux données, le système a toute latitude pour déterminer ces derniers et les combiner de manière à obtenir les meilleures performances.

Nous avons une requête, exprimée en SQL, soumise au système. Comme vous le savez, SQL permet de déclarer un besoin, mais ne dit pas comment calculer le résultat. C'est au système de produire une forme operatoire, un programme, pour effectuer ce calcul. Notez que cette approche a un double avantage. Pour l'utilisateur, elle permet de ne pas se soucier d'algorithmique d'exécution. Pour le système elle laisse la liberté du choix de la meilleure méthode. C'est ce qui fonde l'optimisation, la liberté de déterminer la manière de répondre a un besoin.

Fig. 4.1 – Les requêtes SQL sont *déclaratives*

En base de données, le programme qui évalue une requête a une forme très particulière. On l'appelle plan d'exécution. Il a la forme d'un arbre constitue d'opérateurs qui échangent des données. Chaque opérateur effectue une tache précise et restreinte : transformation, filtrage, combinaisons diverses. Comme nous le verrons, un petit nombre d'opérateurs suffit a évaluer des requêtes, même très complexes. Cela permet au système de construire très rapidement, a la volée, un plan et de commencer a l'exécuter. La question suivante est d'étudier comment le système passe de la requête au plan.

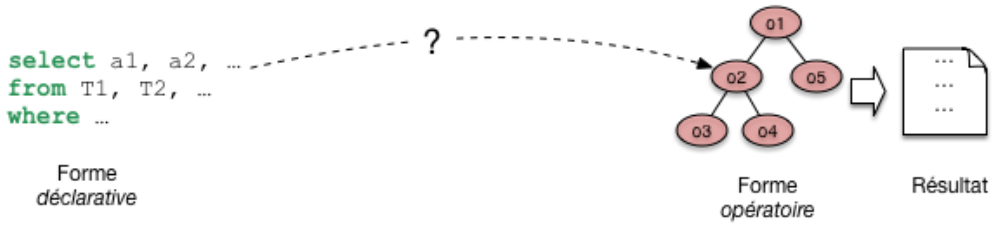

Fig. 4.2 – De la requête SQL au plan d'exécution.

Le passage de SQL a un plan s'effectue en deux étapes, que j'appellerai a et b. Dans l'étape a on tire partie de l'équivalence entre SQL, ou une grande partie de SQL, avec l'algèbre. Pour toute requêtes on peut donc produire une expression de l'algèbre. Et ici on trouve déjà une forme opérationnelle, qui nous dit quelles opérations effectuer. Nous l'appellerons plan d'execution logique. Une expression de l'algèbre peut se représenter comme un arbre, et nous sommes déjà proche d'un n plan d'exécution. Il reste assez abstrait.

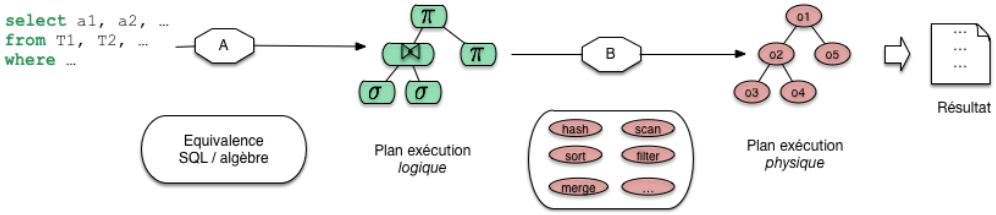

Fig. 4.3 – Les deux phases de l'optimisation

Ce n'est pas tout a fait suffisant. Dans l'étape b le système va choisir des opérateurs particulière, en fonction d'un contexte spécifique. Ce peut être là présence ou non d'index, la taille des tables, la mémoire disponible. Cette étape b donne un plan d'exécution physique, applicable au contexte.

Reste la question de l'optimisation. Il faut ici élargir le schéma : a étape, a ou b, plusieurs options sont possibles. Pour l'étape a, c'est la possibilité d'obtenir plusieurs expressions équivalentes. La figure montre par exemple deux combinaisons possibles issues de la même requête sql. Pour l'étape les options sont liées au choix de l'algorithmique, des opérateurs as exécuter.

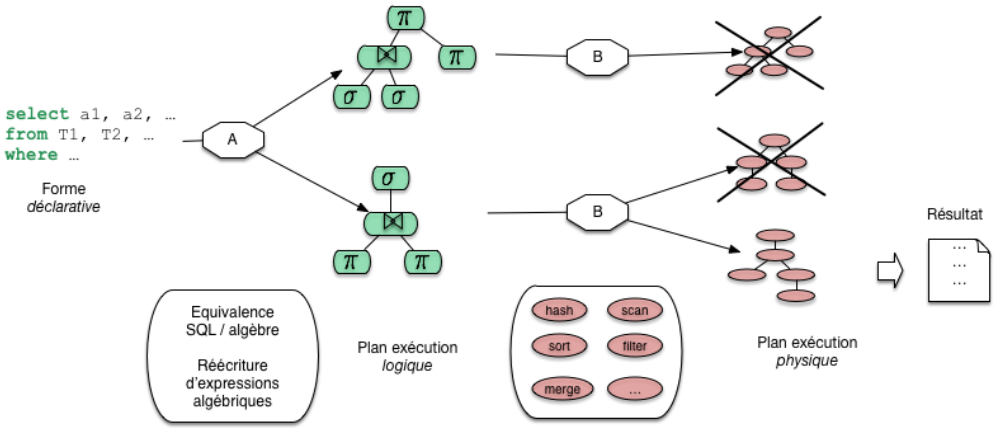

Fig. 4.4 – Processus général d'optimisation et d'évaluation

Cette figure nous donne la perspective générale de cette partie du cours. Nous allons étudier les opérateurs,

les plans d'exécution, les transformations depuis une requête SQL, et quelques critères de choix pour l'optimisation.

4.5 Exercices

Pour varier les exemples, nous utilisons la base (fictive et simplifiée bien entendu) d'un syndic de gestion d'immeuble. Voici son schéma

- Immeuble (**id**, nom, adresse)
- Appart (**id** , no , surface , niveau , *idImmeuble*)
- Personne (**id**, prénom , nom , profession , *idAppart*)
- Propriétaire (**idPersonne , idAppart**, quotePart)

Ce schéma et cette base sont fournis respectivement dans les scripts SchemaImmeuble.sql et BaseImmeuble.sql. Vous pouvez les installer localement si vous le souhaitez. La base est également disponible *via* notre interface en ligne si vous souhaitez effectuer réellement les requêtes proposées parallèlement à votre lecture.

La table Immeuble

Voici le contenu de la table *Immeuble*.

id	nom	adresse
1	Koudalou	3 rue des Martyrs
2	Barabas	2 allée du Grand Turc

La table Appart

Voici le contenu de la table *Appart*.

id	no	surface	niveau	idImmeuble
100	1	150	14	1
101	34	50	15	1
102	51	200	2	1
103	52	50	5	1
104	43	75	3	1
200	1	150	0	2
201	2	250	1	2
202	3	250	2	2

La table Personne

Voici le contenu de la table *Personne*.

id	prénom	nom	profession	idAppart
1		Prof	Enseignant	202
2	Alice	Grincheux	Cadre	103
3	Léonie	Atchoum	Stagiaire	100
4	Barnabé	Simplet	Acteur	102
5	Alphonsine	Joyeux	Rentier	201
6	Brandon	Timide	Rentier	104
7	Don-Jean	Dormeur	Musicien	200

La table Propriétaire

Voici le contenu de la table *Propriétaire*.

idPersonne	idAppart	quotePart
1	100	33
5	100	67
1	101	100
5	102	100
1	202	100
5	201	100
2	103	100

Exercice Ex-alg-1 : gestion des doublons

Dans le modèle théorique, une relation est un ensemble, et ne contient donc aucun doublon. Dans la norme SQL, les choses sont différentes : pour des raisons pratiques, les doublons sont acceptés (les relations sont donc des multiensembles, ou *bag* en anglais) et on doit toujours se demander en exprimant une requête SQL s'il faut les éliminer avec `distinct`.

Posons-nous la question pour le produit cartésien. Soit deux relations `R(idR, A, B, C, ...)` et `S(idS, U, V, W, ..)` avec pour clés primaires `idR` et `idS`. On veut montrer que le `distinct` est-il toujours inutile dans la requête :

```
select *
from R cross join S
```

On montre que le `distinct` est inutile :

— Montrer l'unicité de la paire constituée des identifiants dans $R \times S$.
— En déduire la propriété cherchée

Exercice Ex-alg-2 : du SQL déclaratif à l'expression algébrique

Un SGBD relationnel reçoit une requête SQL, en principe sous forme déclarative, et la traduit alors en expression algébrique, qui donne les opérations à effectuer. À vous de faire le travail : donnez les expressions algébriques équivalentes aux requêtes SQL ci-dessous.

Vous n'avez pas droit aux conjonctions ou disjonctions dans la formule de sélection. Utilisez la composition et l'union.

```
 select t.code, t.nom, t.type
 from Logement as t
 where t.lieu = 'Corse'

 select t.code, t.nom
 from Logement as t
 where t.type = 'Hôtel' and (t.lieu = 'Alpes' or t.capacité >= 100)

select l.code, l.nom
from Logement as l, Activité as a
where l.code = a.codeLogement
and   a.codeActivité = 'Ski'

select distinct l1.nom as nom1, l2.nom as nom2
from Logement as l1, Logement as l2
where l1.type = l2.type

select distinct v.prénom, v.nom
from Voyageur as v
where exists (select ''
       from Séjour as s, Logement as l
       where v. idVoyageur=s.idVoyageur
       and   s.codeLogement = l .code
       and   l.lieu = 'Alpes')

select distinct l.nom
from Logement as l
where not exists (select ''
          from Activité as a
          where l.code = a.codeLogement
          and a.codeActivité = 'Ski')
```

Exercice Ex-alg-3 : de l'algèbre à SQL algébrique

L'atelier consiste à étudier un ensemble de requêtes algébriques, à exprimer leur signification en bon français, et à donner leur formulation en SQL, *forme algébrique*.

Note : Le `except` n'existant pas dans MySQL, vous pouvez exprimer la différence en SQL avec `no exists` comme nous l'avons vu dans le chapitre précédent.

Vous pouvez alors effectuer cette requête en ligne et vérifier le résultat.

Note : Pour simplifier un peu les expressions, on considère ci-dessous que dans la syntaxe $R \bowtie_{A=B} S$, `A` est toujours un attribut de `R` et `B` un attribut de `S`. Une requête comme $R \bowtie_{id=id} S$ ne présente donc pas d'ambiguité.

Voici donc la liste des expressions algébriques à exprimer en SQL (forme algébrique).

- $\pi_{nom,profession}(Personne)$
- $\pi_{idImmeuble,id}(\sigma_{surface>150}(Appart))$
- $\sigma_{no=niveau}(Appart)$
- $\pi_{nom,no,surface}(Immeuble \bowtie_{id=idImmeuble} Appart)$
- $\pi_{nom,no,surface}(Appart \bowtie_{id=idAppart} Personne)$
- $\pi_{nom,idAppart}(Propritaire \bowtie_{idPersonne=id \wedge idAppart=idAppart} Personne))$
- $\pi_{nom,nomI,no,surface}(\rho_{id \to idI, nom \to nomI}(Immeuble) \bowtie_{idI=idImmeuble} (Appart \bowtie_{id=idAppart} Personne))$
- $\rho_{id \to idAppart}(\pi_{id}(Appart)) - \pi_{idAppart}(Personne)$

Exercice Ex-alg-4 : expressions algébriques

Exprimez les requêtes suivantes, en algèbre relationnelle. Vous avez le droit de décomposer une expression complexe en plusieurs étapes, en donnant un nom à chaque étape.

Par exemple : je cherche le nom de l'occupant de l'appartement numéro 51 dans le Koudalou.

Je dois interroger la table Personne (occupant), Appart (numéro 51) et Immeuble (le Koudalou). Voici une décomposition détaillée.

Tout d'abord je sélectionne le Koudalou, dans une relation temporaire `K`.

$$K := \rho_{id \to idI}(\pi_{id}(\sigma_{nom=Koudalou}(Immeuble)))$$

Je prends ensuite l'appartement 51 par jointure entre `Appart` et `K`. J'en profite pour ne conserver, par projection, que les attributs qui m'intéressent, avec parfois un renommage, afin d'éviter de futures ambiguités.

$$A51 := \pi_{id51,surface}(\rho_{id \to id51}(\sigma_{no=51}(Appart) \bowtie_{idImmeuble=idI} K))$$

Il reste à joindre `A51` avec la table `Personne`.

$$\pi_{nom,surface}(Personne \bowtie_{idAppart=id51} A51)$$

En assemblant le tout on aurait l'expression complète. Si vous essayez d'exprimer cela en SQL, avec les opérateurs ensemblistes, vous devriez être convaincus que la forme déclarative est beaucoup plus claire et économique.

Requêtes

- Qui habite le Koudalou ? Vous pouvez décomposer en
  - Les appartements du Koudalou (identifiant)
  - Occupants de ces appartements
- Profession des occupants d'un appartement de plus de 100 m2
- Nom des immeubles ayant un appartement de plus de 150 m2.

— Qui sont les propriétaires de l'appartement de Atchoum ?
— Dans quels immeubles habite un acteur ?
— Qui habite un appartement de moins de 70 m2
— Qui est, au moins partiellement, propriétaire de l'appartement qu'il occupe ?
— Qui occupe un appartement possédé par Prof
— Qui n'est pas propriétaire ?
— Paires de personnes habitant, dans le même immeuble, un appartement de même superficie.
— Dans quels immeubles ne trouve-t-on aucun musicien ?
— Qui possède un appartement sans l'occuper ?

Si vous avez des soucis pour utiliser les lettres grecques, il est possible de les noter en toutes lettres : PI, RHO, SIGMA, CROSS, JOIN, UNION, MINUS.

Exercice Ex-alg-5 : réécriture d'expressions algébriques équivalentes

On dispose de deux tables $T_1(A, B, C)$ et $T_2(D, E, F)$. Donnez une expression algébrique équivalente à la suivante, dans laquelle on utilise la jointure mais pas le produit cartésien, et où les sélections s'appliquent directement aux tables.

$$\sigma_{A=C \wedge C>D \wedge E=F}(T_1 \times T_2)$$

CHAPITRE 5

SQL, récapitulatif

Ce chapitre présente les compléments du langage d'interrogation SQL (la partie dite *Langage de Manipulation de Données* ou LMD) dans le cadre d'un récapitulatif. Ces compléments présentent peu de difficulté dans la mesure où la véritable complexité réside d'une part dans l'interprétation des requêtes complexes qui font parfois appel à des logiques sophistiquées et d'autre part dans la multiplicité des variantes syntaxiques qui peut parfois troubler.

Les deux chapitres précédents devraient avoir réglé ces problèmes d'interprétation. Vous savez maintenant que SQL propose deux paradigmes d'interrogation, l'un déclaratif et l'autre procédural. Les requêtes se comprennent soit via leur équivalent en formules logiques, soit en les considérant comme des opérations ensemblistes.

Dans ce chapitre nous utilisons systématiquement l'approche déclarative. Vous pouvez les reformuler sous leur forme ensembliste si vous le souhaitez.

La base prise comme exemple dans ce chapitre est celle des immeubles.

5.1 S1 : le bloc `select-from-where`

Supports complémentaires :

— Diapositives: le bloc `select-from-where

— Vidéo sur le bloc `select-from-where

Dans cette session, nous étudions les compléments à la forme de base d'une requête SQL, que nous appelons *bloc*, résumée ainsi :

```
select liste_expressions
from   relations_sources
[where liste_conditions]
[order by critère_de_tri]
```

Parmi les quatre clauses `select`, `form`, `where` et `order by`, les deux dernières sont optionnelles. La recherche la plus simple consiste à récupérer le contenu complet d'une table. On n'utilise pas la clause `where` et le * désigne tous les attributs.

```
select * from Immeuble
```

id	nom	adresse
1	Koudalou	3 rue des Martyrs
2	Barabas	2 allée du Grand Turc

L'ordre des trois clauses `select from` et `where` est trompeur pour la signification d'une requête. Comme nous l'avons déjà détaillé dans les chapitres qui précédent l'inteprétation s'effectue *toujours* de la manière suivante :

- la clause `from` définit l'espace de recherche en fonction d'un ensemble de sources de données ;
- la clause `where` exprime un ensemble de conditions sur la source : seuls les nuplets pour lesquels ces conditions sont satisfaites sont conservés ;
- enfin la clause `select` construit un nuplet-résultat grâce à une liste d'expressions appliquées aux nuplets de la source ayant passé le filtre du `where`.

5.1.1 La clause `from`

L'espace de recherche est défini dans la clause `from` par une ou plusieurs tables. Par « table » il ne faut pas ici comprendre forcément « une des tables de la base » courante même si c'est le cas le plus souvent rencontré. SQL est beaucoup général que cela : une table dans un `from` peut également être *résultat* d'une autre requête. On parlera de table *basée* et de table *calculée* pour distinguer ces deux cas. Ce peut également être une table stockée dans une autre base ou une table calculée à partir de tables basées dans plusieurs bases ou une combinaison de tout cela.

Voici une première requête qui ramène les immeubles dont l'id vaut 1.

```
select nom, adresse
from Immeuble
where id=1
```

Il n'aura pas échappé au lecteur attentif que le résultat est lui-même une table (calculée et non basée). Pourquoi ne pourrait-on pas interroger cette table calculée comme une autre ? C'est possible en SQL comme le montre l'exemple suivant :

```
select *
from (select nom, adresse from Immeuble where id=1) as Koudalou
```

On a donc placé une requête SQL dans le `from` où elle définit un espace de recherche constitué de son propre résultat. Le mot-clé `as` permet de donner un nom temporaire au résultat. En d'autres termes `Koudalou` est le nom de la table calculée sur laquelle s'effectue la requête. Cette table temporaire n'existe que pendant l'exécution.

Note : Comme nous l'avons vu, cette approche est de nature algébrique : on manipule dans le `from` des ensembles, stockés (les tables) ou calculés (obtenus par des requêtes). C'est une syntaxe en plus pour dire la même chose, donc on peut très bien se passer de la seconde formulation. Il est plus intéressant de prolonger cette idée d'interroger une relation *calculée* en donnant définivement un nom à la requête qui sélectionne l'immeuble. En SQL cela s'appelle une *vue*. On crée une vue dans un schéma avec la commande `create view`.

```
create view Koudalou as
   select nom, adresse from Immeuble where id=1
```

Une fois créée une vue peut être utilisée comme espace de recherche exactement comme une table basée. Le fait que son contenu soit calculé reste transparent pour l'utilisateur.

```
select nom, adresse from Koudalou
```

Les vues sont traitées en détail dans le chapitre consacré aux schémas relationnels.

L'interprétation du `from` est indépendante de l'origine des tables : tables basées, tables calculées, et vues. Comme nous l'avons vu dans les chapitres précédents, il existe deux manières de spécifier l'espace de recherche avec le `from`. La première est la forme déclarative dans laquelle on sépare le nom des tables par des virgules.

```
select * from Immeuble as i, Appart as a
```

Dans ce cas, le nom d'une table sert à définir une variable nuplet (voir chapitre *SQL, langage déclaratif*) à laquelle on peut affecter tous les nuplets de la table. Les variables peuvent être explicitement nommées avec la mot-clé `as` (elles s'appellent `i` et `a` dans la requête ci-dessus). On peut aussi omettre le `as`, dans ce cas le nom de la variable est (implcitement) le nom de la table.

```
select * from Immeuble, Appart
```

Un cas où le `as` est obligatoire est l'auto-jointure : on veut désigner deux nuplets de la même table. Exemple : on veut les paires d'appartement du même immeuble.

```
select a1.no, a2.no
from Appart as a1, Appart as a2
where a1.idImmeuble = a2.idImmeuble
```

En l'absence du `as` et de l'utilisation du nom de la variable comme préfixe, il y aurait ambiguité sur le nom des attributs.

La deuxième forme du `from` définit l'espace de recherche par une opération algébrique, jointure ou produit cartésien.

```
select * from Immeuble cross join Appart
```

Cette formulation revient à définir une table virtuelle (appelons-là *Tfrom*) qui tient lieu d'espace de recherche par la suite. L'affichage ci-dessus nous montre quel est l'espace de recherche *Tfrom* de la requête précédente.

id	nom	adresse	id	sur-face	ni-veau	idIm-meuble	no
1	Koudalou	3 rue des Martyrs	100	150	14	1	1
2	Barabas	2 allée du Grand Turc	100	150	14	1	1
1	Koudalou	3 rue des Martyrs	101	50	15	1	34
2	Barabas	2 allée du Grand Turc	101	50	15	1	34
1	Koudalou	3 rue des Martyrs	102	200	2	1	51
2	Barabas	2 allée du Grand Turc	102	200	2	1	51
1	Koudalou	3 rue des Martyrs	103	50	5	1	52
2	Barabas	2 allée du Grand Turc	104	75	3	1	43
1	Koudalou	3 rue des Martyrs	104	75	3	1	43
2	Barabas	2 allée du Grand Turc	103	50	5	1	52
1	Koudalou	3 rue des Martyrs	200	150	0	2	1
2	Barabas	2 allée du Grand Turc	200	150	0	2	1
1	Koudalou	3 rue des Martyrs	201	250	1	2	1
2	Barabas	2 allée du Grand Turc	201	250	1	2	1
1	Koudalou	3 rue des Martyrs	202	250	2	2	2
2	Barabas	2 allée du Grand Turc	202	250	2	2	2

La clause de jointure `join` définit un espace de recherche constitué des paires de nuplets pour lesquels la condition de jointure est vraie.

```
select *
from Immeuble join Appart on (Immeuble.id=Appart.idImmeuble)
```

On obtient le résultat suivant.

id	nom	adresse	id	sur-face	ni-veau	idIm-meuble	no
1	Koudalou	3 rue des Martyrs	100	150	14	1	1
1	Koudalou	3 rue des Martyrs	101	50	15	1	34
1	Koudalou	3 rue des Martyrs	102	200	2	1	51
1	Koudalou	3 rue des Martyrs	103	50	5	1	52
1	Koudalou	3 rue des Martyrs	104	75	3	1	43
2	Barabas	2 allée du Grand Turc	200	150	0	2	1
2	Barabas	2 allée du Grand Turc	201	250	1	2	1
2	Barabas	2 allée du Grand Turc	202	250	2	2	2

Dernière précision au sujet du `from` : l'ordre dans lequel on énumère les tables n'a aucune importance.

5.1.2 La clause `where`

La clause `where` permet d'exprimer des conditions portant sur les nuplets désignés par la clause `from`. Ces conditions suivent en général la syntaxe `expr1 [not]` Θ `expr2`, où `expr1` et `expr2` sont deux expressions construites à partir de noms d'attributs, de constantes et de fonctions, et Θ est l'un des opérateurs de comparaison classique < > <= >= != =.

Les conditions se combinent avec les connecteurs booléens `and` `or` et `not`. SQL propose également un prédicat `in` qui teste l'appartenance d'une valeur à un ensemble. Il s'agit (du moins tant qu'on n'utilise pas les requêtes imbriquées) d'une facilité d'écriture pour remplacer le `or`. La requête

```
select *
from Personne
where profession='Acteur'
or profession='Rentier'
```

s'écrit de manière équivalente avec un `in` comme suit :

```
select *
from Personne
where profession in ('Acteur', 'Rentier')
```

id	prénom	nom	profession	idAppart
4	Barnabé	Simplet	Acteur	102
5	Alphonsine	Joyeux	Rentier	201

Pour les chaînes de caractères, SQL propose l'opérateur de comparaison `like`, avec deux caractères de substitution :

— le « % » remplace n'importe quelle sous-chaîne ;
— le « _ » remplace n'importe quel caractère.

L'expression `_ou%ou` est donc interprétée par le `like` comme toute chaîne commençant par un caractère suivi de « ou » suivi de n'importe quelle chaîne suivie une nouvelle fois de « ou ».

```
select *
from Immeuble
where nom like '_ou%ou'
```

id	nom	adresse
1	Koudalou	3 rue des Martyrs

Il est également possible d'exprimer des conditions sur des tables calculées par d'autre requêtes SQL incluses dans la clause `where` et habituellement désignées par le terme de « requêtes imbriquées ». On pourra par exemple demander la liste des personnes dont l'appartement fait partie de la table calculée des appartements situés au-dessus du troisième niveau.

```
select * from Personne
where idAppart in (select id from Appart where niveau > 3)
```

id	prénom	nom	profession	idAppart
2	Alice	Grincheux	Cadre	103
3	Léonie	Atchoum	Stagiaire	100

Avec les requêtes imbriquées on entre dans le monde incertain des requêtes qui semblent claires mais finissent par ne plus l'être du tout. La difficulté vient souvent du fait qu'il faut raisonner simultanément sur plusieurs requêtes qui, de plus, sont souvent interdépendantes (les données sélectionnées dans l'une servent de paramètre à l'autre). Il est très souvent possible d'éviter les requêtes imbriquées comme nous l'expliquons dans ce chapitre.

5.1.3 Valeurs manquantes : le `null`

En théorie, dans une table relationnelle, tous les attributs ont une valeur. En pratique, certaines valeurs peuvent être inconnues ou manquantes : on dit qu'elles sont à `null`. Le `null` *n'est pas une valeur spéciale*, c'est une absence de valeur.

Note : Les valeurs à `null` sont une source de problème, car elles rendent parfois le résultat des requêtes difficile à comprendre. Mieux vaut les éviter si c'est possible.

Il est impossible de déterminer quoi que ce soit à partir d'une valeur à `null`. Dans le cas des comparaisons, la présence d'un `null` renvoie un résultat qui n'est ni `true` ni `false` mais `unknown`, une valeur booléenne intermédiaire. Reprenons à nouveau la table *Personne* avec un des prénoms à `null`. La requête suivante devrait ramener tous les nuplets.

```
select *
from Personne
where prénom like '%'
```

Mais la présence d'un `null` empêche l'inclusion du nuplet correspondant dans le résultat.

id	prénom	nom	profession	idAppart
2	Alice	Grincheux	Cadre	103
3	Léonie	Atchoum	Stagiaire	100
4	Barnabé	Simplet	Acteur	102
5	Alphonsine	Joyeux	Rentier	201
6	Brandon	Timide	Rentier	104
7	Don-Jean	Dormeur	Musicien	200

Cependant la condition `like` n'a pas été évaluée à `false` comme le montre la requête suivante.

```
select  *
from Personne
where prénom not like  '%'
```

On obtient un résultat vide, ce qui montre bien que le `like` appliqué à un `null` ne renvoie pas `false` (car sinon on aurait `not false = true`). C'est d'ailleurs tout à fait normal puisqu'il n'y a aucune raison de dire qu'une absence de valeur ressemble à n'importe quelle chaîne.

Les tables de vérité de la logique trivaluée de SQL sont définies de la manière suivante. Tout d'abord on affecte une valeur aux trois constantes logiques :

- `true` vaut 1
- `false` vaut 0
- `unknown` vaut 0.5

Les connecteurs booléens s'interprètent alors ainsi :

- `val1 and val2`, = min(val1 val2)
- `val1 or val2` = max(val1, val2)
- `not val1` = 1 - val1.

On peut vérifier notamment que `not unknown` vaut toujours `unknown`. Ces définitions sont claires et cohérentes. Cela étant il faut mieux prévenir de mauvaises surprises avec les valeurs à `null`, soit en les interdisant à la création de la table avec les options `not null` ou `default`, soit en utilisant le test `is null` (ou son complément `is not null`). La requête ci-dessous ramène tous les nuplets de la table, même en présence de `null`.

```
select *
from Personne
where prénom like '%'
or prénom is null
```

id	prénom	nom	profession	idAppart
1		Prof	Enseignant	202
2	Alice	Grincheux	Cadre	103
3	Léonie	Atchoum	Stagiaire	100
4	Barnabé	Simplet	Acteur	102
5	Alphonsine	Joyeux	Rentier	201
6	Brandon	Timide	Rentier	104
7	Don-Jean	Dormeur	Musicien	200

Attention le test `valeur = null` n'a pas de sens. On ne peut pas être égal à une absence de valeur.

5.1.4 La clause `select`

Finalement, une fois obtenus les nuplets du `from` qui satisfont le `where` on crée à partir de ces nuplets le résultat final avec les expressions du `select`.

Si on indique explicitement les attributs au lieu d'utiliser *, leur nombre détermine le nombre de colonnes de la table calculée. Le nom de chaque attribut dans cette table est par défaut l'expression du `select` mais on peut indiquer explicitement ce nom avec `as`. Voici un exemple qui illustre également une fonction assez utile, la concaténation de chaînes.

```
select concat(prénom, ' ', nom) as 'nomComplet'
from Personne
```

nomComplet
null
Alice Grincheux
Léonie Atchoum
Barnabé Simplet
Alphonsine Joyeux
Brandon Timide
Don-Jean Dormeur

Note : La fonction `concat()` ici utilisée est spécifique à MySQL.

Le résultat montre que l'une des valeurs est à `null`. Logiquement toute opération appliquée à un `null` renvoie un `null` en sortie puisqu'on ne peut calculer aucun résultat à partir d'une valeur inconnue. Ici c'est le prénom de l'une des personnes qui manque. La concaténation du prénom avec le nom est une opération qui « propage » cette valeur à `null`. Dans ce cas, il faut utiliser une fonction (spécifique à chaque système)à qui remplace la valeur à `null` par une valeur de remplacement. Voici la version MySQL (fonction `ifnull(attribut, remplacement)`).

```
select concat(ifnull(prénom,' '), ' ', nom) as 'nomComplet'
from Personne
```

Une « expression » dans la clause `select` désigne ici, comme dans tout langage, une construction syntaxique qui prend une ou plusieurs valeurs en entrée et produit une valeur en sortie. Dans sa forme la plus simple, une expression est simplement un nom d'attribut ou une constante comme dans l'exemple suivant.

```
select surface, niveau, 18 as 'EurosParm2'
from Appart
```

surface	niveau	EurosParm2
150	14	18
50	15	18
200	2	18
50	5	18
75	3	18
150	0	18
250	1	18
250	2	18

Les attributs `surface` et `niveau` proviennent de *Appart* alors que 18 est une constante qui sera répétée autant de fois qu'il y a de nuplets dans le résultat. De plus, on peut donner un nom à cette colonne avec la commande `as`. Voici un second exemple qui montre une expression plus complexe. L'utilisateur (certainement un agent immobilier avisé et connaissant bien SQL) calcule le loyer d'un appartement en fonction d'une savante formule qui fait intervenir la surface et le niveau.

```
select no, surface, niveau,
        (surface * 18) * (1 + (0.03 * niveau)) as loyer
from Appart
```

no	surface	niveau	loyer
1	150	14	3834.00
34	50	15	1305.00
51	200	2	3816.00
52	50	5	1035.00
1	250	1	4635.00
2	250	2	4770.00

SQL fournit de très nombreux opérateurs et fonctions de toute sorte qui sont clairement énumérées dans la documentation de chaque système. Elles sont particulièrement utiles pour des types de données un peu délicat à manipuler comme les dates.

Une extension rarement utilisée consiste à effectuer des tests sur la valeur des attributs à l'intérieur de la clause `select` avec l'expression `case` dont la syntaxe est :

```
case
  when test then expression
  [when ...]
  else  expression
end
```

Ces tests peuvent être utilisés par exemple pour effectuer un *décodage* des valeurs quand celles-ci sont difficiles à interpréter ou quand on souhaite leur donner une signification dérivée. La requête ci-dessous classe les appartements en trois catégories selon la surface.

```
 select no, niveau, surface,
       case when surface <= 50 then 'Petit'
            when surface > 50 and surface <= 100 then 'Moyen'
            else 'Grand'
       end as categorie
from Appart
```

no	niveau	surface	categorie
1	14	150	Grand
34	15	50	Petit
51	2	200	Grand
52	5	50	Petit
43	3	75	Moyen
10	0	150	Grand
1	1	250	Grand
2	2	250	Grand

5.1.5 Jointure interne, jointure externe

La jointure est une opération indispensable dès que l'on souhaite combiner des données réparties dans plusieurs tables. Nous avons déjà étudié en détail la conception et l'expression des jointures. On va se contenter ici de montrer quelques exemples en forme de récapitulatif, sur notre base d'immeubles.

Note : Il existe beaucoup de manières différentes d'exprimer les jointures en SQL. Il est recommandé de se limiter à la forme de base donnée ci-dessous qui est plus facile à interpréter et se généralise à un nombre de tables quelconques.

Jointure interne

Prenons l'exemple d'une requête cherchant la surface et le niveau de l'appartement de M. Barnabé Simplet.

```
select p.nom, p.prénom,  a.surface, a.niveau
from Personne as p, Appart as a
where prénom='Barnabé'
and nom='Simplet'
and   a.id = p.idAppart
```

nom	prénom	surface	niveau
Simplet	Barnabé	200	2

Une première difficulté à résoudre quand on utilise plusieurs tables est la possibilité d'avoir des attributs de même nom dans l'union des schémas, ce qui soulève des ambiguités dans les clauses `where` et `select`. On

résout cette ambiguité en préfixant les attributs par le nom des variables-nuplet dont ils proviennent.

Notez que la levée de l'ambiguité en préfixant par le nom de la variable-nuplet n'est nécessaire que pour les attributs qui apparaissent en double soit ici `id` qui peut désigner l'identifiant de la personne ou celui de l'appartement.

Comme dans la très grande majorité des cas la jointure consiste à exprimer une égalité entre la clé primaire de l'une des tables et la clé étrangère correspondante de l'autre. Mais rien n'empêche d'exprimer des conditions de jointure sur n'importe quel attribut et pas seulement sur ceux qui sont des clés.

Imaginons que l'on veuille trouver les appartements d'un même immeuble qui ont la même surface. On veut associer un nuplet de *Appart* à un autr nuplet de *Appart* avec les conditions suivantes :

— ils sont dans le même immeuble (attribut `idImmeuble`) ;
— ils ont la même valeur pour l'attribut `surface` ;
— ils correspondent à des appartements distincts (attributs `id`).

La requête exprimant ces conditions est donc :

```
select a1.id as idAppart1, a1.surface as surface1, a1.niveau as niveau1,
       a2.id as idAppart2, a2.surface as surface2, a2.niveau as niveau2
from Appart a1, Appart a2
where a1.id != a2.id
and a1.surface = a2.surface
and a1.idImmeuble = a2.idImmeuble
```

Ce qui donne le résultat suivant :

idAppart1	surface1	niveau1	idAppart2	surface2	niveau2
103	50	5	101	50	15
101	50	15	103	50	5
202	250	2	201	250	1
201	250	1	202	250	2

On peut noter que dans le résultat la même paire apparaît deux fois avec des ordres inversés. On peut éliminer cette redondance en remplaçant `a1.id != a2.id` par `a1.id < a2.id`.

Voici quelques exemples complémentaires de jointure.

— Qui habite un appartement de plus de 200 m2 ?

```
select prénom, nom, profession
from Personne, Appart
where idAppart = Appart.id
and  surface >= 200
```

Attention à lever l'ambiguité sur les noms d'attributs quand ils peuvent provenir de deux tables (c'est le cas ici pour `id`).

— Qui habite le Barabas ?

```
select prénom, p.nom, no, surface, niveau
from   Personne as p, Appart as a, Immeuble as i
where  p.idAppart=a.id
and    a.idImmeuble=i.id
and    i.nom='Barabas'
```

— Qui habite un appartement qu'il possède et avec quelle quote-part ?

```
select prénom, nom, quotePart
from   Personne as p, Propriétaire as p2, Appart as a
where  p.id=p2.idPersonne /* p est propriétaire */
and    p2.idAppart=a.id   /* de l'appartement a */
and    p.idAppart=a.id   /* et il y habite     */
```

— De quel(s) appartement(s) Alice Grincheux est-elle propriétaire et dans quel immeuble ? Voici la requête sur les quatre tables avec des commentaires inclus montrant les jointures.

```
 select i.nom, no, niveau, surface
from  Personne as p, Appart as a, Immeuble as i, Propriétaire as p2
where  p.id=p2.idPersonne /* Jointure PersonnePropriétaire */
and    p2.idAppart = a.id /* Jointure PropriétaireAppart */
and    a.idImmeuble= i.id /* Jointure AppartImmeuble */
and    p.nom='Grincheux' and p.prénom='Alice'
```

Attention à lever l'ambiguité sur les noms d'attributs quand ils peuvent provenir de deux tables (c'est le cas ici pour `id`).

L'approche déclarative d'expression des jointures est une manière tout à fait recommandable de procéder surtout pour les débutants SQL. Elle permet de se ramener toujours à la même méthode d'interprétation et consolide la compréhension des principes d'interrogation d'une base relationnelle.

Toutes ces jointures peuvent s'exprimer avec d'autres syntaxes : tables calculées dans le `from` opérateur de jointure dans le `from` ou (pas toujours) requêtes imbriquées. À l'exception notable des jointures externes, elles n'apportent aucune expressivité supplémentaire. Toutes ces variantes constituent des moyens plus ou moins commodes d'exprimer différemment la jointure.

Jointure externe

Qu'est-ce qu'une jointure externe ? Effectuons la requête qui affiche tous les appartements avec leur occupant.

```
select idImmeuble, no, niveau, surface, nom, prénom
from  Appart as a, Personne as p
where  p.idAppart=a.id
```

Voici ce que l'on obtient :

idImmeuble	no	niveau	surface	nom	prénom
2	2	2	250	Prof	null
1	52	5	50	Grincheux	Alice
1	1	14	150	Atchoum	Léonie
1	51	2	200	Simplet	Barnabé
2	1	1	250	Joyeux	Alphonsine
1	43	3	75	Timide	Brandon
2	10	0	150	Dormeur	Don-Jean

Il manque un appartement, le 34 du Koudalou. En effet cet appartement n'a pas d'occupant. Il n'y a donc aucune possibilité que la condition de jointure soit satisfaite.

La jointure externe permet d'éviter cette élimination parfois indésirable. On considère alors une hiérarchie entre les deux tables. La première table (en général celle de gauche) est dite « directrice » et tous ses nuplets, même ceux qui ne trouvent pas de correspondant dans la table de droite, seront prises en compte. Les nuplets de la table de droite sont en revanche optionnels.

Si pour un nuplet de la table de gauche on trouve un nuplet satisfaisant le critère de jointure dans la table de droite, alors la jointure s'effectue normalement. Sinon, les attributs provenant de la table de droite sont affichés à `null`. Voici la jointure externe entre *Appart* et *Personne*. Le mot-clé `left` est optionnel.

```
select idImmeuble, no niveau, surface, nom, prénom
from  Appart as a left outer join Personne as p on (p.idAppart=a.id)
```

idImmeuble	no	niveau	surface	nom	prénom
1	1	14	150	Atchoum	Rachel
1	34	15	50	null	null
1	51	2	200	Simplet	Barnabé
1	52	5	50	Grincheux	Alice
2	1	1	250	Joyeux	Alphonsine
2	2	2	250	Prof	null

Notez les deux attributs `prénom` et `nom` à `null` pour l'appartement 34.

Il existe un `right outer join` qui prend la table de droite comme table directrice. On peut combiner la jointure externe avec des jointures normales des sélections des tris etc. Voici la requête qui affiche le nom de l'immeuble en plus des informations précédentes et trie par numéro d'immeuble et numéro d'appartement.

```
select i.nom as nomImmeuble, no, niveau, surface, p.nom as nomPersonne, prénom
from  Immeuble  as i
       join
           (Appart as a left outer join Personne as p
                       on (p.idAppart=a.id))
        on (i.id=a.idImmeuble)
order by i.id, a.no
```

5.1.6 Tri et élimination de doublons

SQL renvoie les nuplets du résultat sans se soucier de la présence de doublons. Si on cherche par exemple les surfaces des appartements avec

```
select surface
from Appart
```

on obtient le résultat suivant.

surface
150
50
200
50
250
250

On a autant de fois une valeur qu'il y a de nuplets dans le résultat intermédiaire après exécution des clauses `from` et `where`. En général, on ne souhaite pas conserver ces nuplets identiques dont la répétition n'apporte aucune information. Le mot-clé `distinct` placé juste après le `select` permet d'éliminer ces doublons.

```
select distinct surface
from Appart
```

surface
150
50
200
250

Le `distinct` est à éviter quand c'est possible car l'élimination des doublons peut entraîner des calculs coûteux. Il faut commencer par calculer entièrement le résultat, puis le trier ou construire une table de hachage, et enfin utiliser la structure temporaire obtenue pour trouver les doublons et les éliminer. Si le résultat est de petite taille cela ne pose pas de problème. Sinon, on risque de constater une grande différence de temps de réponse entre une requête sans `distinct` et la même avec `distinct`.

On peut demander explicitement le tri du résultat sur une ou plusieurs expressions avec la clause `order by` qui vient toujours à la fin d'une requête `select`. La requête suivante trie les appartements par surface puis, pour ceux de surface identique, par niveau.

```
select *
from Appart
order by surface, niveau
```

id	surface	niveau	idImmeuble	no
103	50	5	1	52
101	50	15	1	34
100	150	14	1	1
102	200	2	1	51
201	250	1	2	1
202	250	2	2	2

Par défaut, le tri est en ordre ascendant. On peut inverser l'ordre de tri d'un attribut avec le mot-clé `desc` .

```
select *
from Appart
order by surface desc,  niveau desc
```

id	surface	niveau	idImmeuble	no
202	250	2	2	2
201	250	1	2	1
102	200	2	1	51
100	150	14	1	1
101	50	15	1	34
103	50	5	1	52

Bien entendu, on peut trier sur des expressions au lieu de trier sur de simples noms d'attribut.

5.1.7 Quiz

5.2 S2 : Requêtes et sous-requêtes

Supports complémentaires :

Pas de support vidéo pour cette session qui ne fait que récapituler les différentes syntaxes équivalentes pour exprimer une même requête. Ne vous laissez pas troubler par la multiplicité des options offertes pqr SQL. En choisissant un dialecte et un seul (vous avez compris que je vous recommande la partie déclarative, logique de SQL) vous pourrez tout exprimer sans avoir à vous poser des questions sans fin. Vos requêtes n'en seront que plus cohérentes et lisibles.

Dans tout ce qui précède, les requêtes étaient « à plat », avec un seul bloc `select-from-where`. SQL est assez riche (ou assez inutilement compliqué, selon les goûts) pour permettre des expressions complexes combinant plusieurs blocs. On a dans ce cas une requête principale, et des sous-requêtes, ou requêtes imbriquées.

Disons-le tout de suite : à l'exception des requêtes avec négation `not exists`, toutes les requêtes imbriquées peuvent s'écrire de manière équivalente à plat, et on peut juger que c'est préférable pour des raisons de lisibilité et de cohérence d'écriture. Cette session essaie en tout cas de clarifier les choses.

5.2.1 Requêtes imbriquées

Reprenons l'exemple de la requête trouvant la surface et le niveau de l'appartement de M. Simplet. On peut l'exprimer avec une requête imbriquée de deux manières. La première est la forme déclarative classique.

```
select surface, niveau
from Appart as a, Personne as p
where p.prénom='Barnabé' and p.nom='Simplet'
and a.id = p.idAppart
```

On remarque qu'aucun attribut de la table `Personne` n'est utilisé pour construire le résultat. On peut donc utiliser une sous-requête (ou requête imbriquée).

```
select surface, niveau
from Appart
where  id  in (select idAppart
               from Personne
               where prénom='Barnabé' and nom='Simplet')
```

Le mot-clé `in` exprime la condition d'appartenance de l'identifiant de l'appartement à l'ensemble d'identifiants constitué avec la requête imbriquée. Il doit y avoir correspondance entre le nombre et le type des attributs auxquels s'applique la comparaison par `in`. L'exemple suivant montre une comparaison entre des paires d'attributs (ici on cherche des informations sur les propriétaires).

```
select prénom, nom, surface, niveau
from   Appart as a, Personne as p
where  a.id = p.idAppart
and    (p.id, p.idAppart)
          in (select idPersonne, idAppart from Propriétaire)
```

prénom	nom	surface	niveau
null	Prof	250	2
Alice	Grincheux	50	5
Alphonsine	Joyeux	250	1

Il est bien entendu assez direct de réécrire la requête ci-dessus comme une jointure classique (exercice). Parfois l'expression avec requête imbriquée peut s'avérer plus naturelle. Supposons que l'on cherche les immeubles dans lesquels on trouve un appartement de 50 m2. Voici l'expression avec requête imbriquée.

```
select *
from Immeuble
where id in (select idImmeuble from Appart where surface=50)
```

id	nom	adresse
1	Koudalou	3 rue des Martyrs

La requête directement réécrite en jointure donne le résultat suivant :

```
select i.*
from   Immeuble as i,Appart as a
where  i.id=a.idImmeuble
and    surface=50
```

id	nom	adresse
1	Koudalou	3 rue des Martyrs
1	Koudalou	3 rue des Martyrs

On obtient deux fois le même immeuble puisqu'il peut être associé à deux appartements différents de 50 m2. Il suffit d'ajouter un `distinct` après le `select` pour régler le problème, mais on peut considérer que dans ce cas la requête imbriquée est plus appropriée. Attention cependant : il n'est pas possible de placer dans le résultat des attributs appartenant aux tables des requêtes imbriquées.

Le principe général des requêtes imbriquées est d'exprimer des conditions sur des tables calculées par des requêtes. Cela revient, dans le cadre formel qui soutient SQL, à appliquer une quantification sur une collection constituée par une requête (voir chapitre *SQL, langage déclaratif*).

Ces conditions sont les suivantes :

- `exists R` : renvoie `true` si R n'est pas vide `false` sinon.
- t `in R` où est un nuplet dont le type (le nombre et le type des attributs) est celui de R : renvoie `true` si t appartient à R `false` sinon.
- v *cmp* `any R` où *cmp* est un comparateur SQL (< > = etc.) : renvoie `true` si la comparaison avec *au moins un* des nuplets de la table R renvoie `true`.
- v *cmp* `all R` où *cmp* est un comparateur SQL (< > = etc.) : renvoie `true` si la comparaison avec *tous* les nuplets de la table R renvoie `true`.

De plus toutes ces expressions peuvent être préfixées par `not` pour obtenir la négation. La richesse des expressions possibles permet d'effectuer une même interrogation en choisissant parmi plusieurs syntaxes possibles. En général, tout ce qui n'est pas basé sur une négation `not in` ou `not exists` peut s'exprimer *sans* requête imbriquée.

Le `all` peut se réécrire avec une négation puisque si une propriété est *toujours* vraie il n'existe pas de cas où elle est fausse. La requête ci-dessous applique le `all` pour chercher le niveau le plus élevé de l'immeuble 1.

```
select * from Appart
    where idImmeuble=1
    and    niveau >= all (select niveau from Appart where idImmeuble=1)
```

Le `all` exprime une comparaison qui vaut pour *toutes* les nuplets ramenés par la requête imbriquée. La formulation avec `any` s'écrit :

```
select * from Appart
    where idImmeuble=1
    and   not (niveau < any (select niveau from Appart where idImmeuble=1))
```

Rien de nouveau du point de vue expressif : on peut prendre les étages tels *qu'il n'existe pas* un niveau supérieur.

```
select * from Appart as a1
    where a1.idImmeuble=1
    and    not exists (select *
                       from Appart as a2
                       where a1.idImmeuble=1 and a1.niveau < a2.niveau)
```

Attention aux valeurs à `null` dans ce genre de situation : toute comparaison avec une de ces valeurs renvoie `unknown` et cela peut entraîner l'échec du `all`. Il n'existe pas d'expression avec jointure qui puisse exprimer ce genre de condition.

5.2.2 Requêtes correlées

Les exemples de requêtes imbriquées donnés précédemment pouvaient être évalués indépendamment de la requête principale, ce qui permet au système (s'il le juge nécessaire) d'exécuter la requête en deux phases. La clause `exists` fournit encore un nouveau moyen d'exprimer les requêtes vues précédemment en basant la sous-requête sur une ou plusieurs valeurs issues de la requête principale. On parle alors de requêtes *correlées*.

Voici encore une fois la recherche de l'appartement de M. Barnabé Simplet exprimée avec `exists` :

```
select * from Appart
where  exists  (select * from Personne
                where  prénom='Barnabé' and nom='Simplet'
                and    Personne.idAppart=Appart.id)
```

On obtient donc une nouvelle technique d'expression qui permet d'aborder le critère de recherche sous une troisième perspective : on conserve un appartement si, *pour cet appartement*, l'occupant s'appelle Barnabé Simplet. Il s'agit assez visiblement d'une jointure mais entre deux tables situées dans des requêtes (ou plutôt des « blocs ») distinctes. La condition de jointure est appelée corrélation d'où le nom de ce type de technique.

Les jointures dans lesquelles le résultat est construit à partir d'une seule table peuvent d'exprimer avec `exists` ou `in`. Voici quelques exemples reprenant des requêtes déjà vues précédemment.

— Qui habite un appartement de plus de 200 m2 ?
Avec `in` :

```
select prénom, nom, profession
from Personne
where idAppart in (select id from Appart where surface >= 200)
```

Avec `exists` :

```
select prénom, nom, profession
from Personne  p
where exists (select * from Appart a
              where a.id=p.idAppart
              and surface >= 200)
```

— Qui habite le Barabas ?
Avec `in` :

```
select prénom, nom, no, surface, niveau
from   Personne as p, Appart as a
where  p.idAppart=a.id
and a.idImmeuble in
         (select id from Immeuble
          where  nom='Barabas')
```

Avec `exists` :

```
select prénom, nom, no, surface, niveau
from Personne as p, Appart as a
where  p.idAppart=a.id
and exists (select * from Immeuble i
            where  i.id=a.idImmeuble
            and i.nom='Barabas')
```

Important : dans une sous-requête associée à la clause `exists` peu importent les attributs du `select` puisque la condition se résume à : cette requête ramène-t-elle au moins un nuplet ou non ? On peut donc systématiquement utiliser `select *` ou `select ''`

Enfin rien n'empêche d'utiliser plusieurs niveaux d'imbrication au prix d'une forte dégradation de la lisibilité. Voici la requête « De quel(s) appartement(s) Alice Grincheux est-elle propriétaire et dans quel immeuble ? » écrite avec plusieurs niveaux.

```
select i.nom, no, niveau, surface
from  Immeuble as i, Appart as a
where  a.idImmeuble= i.id
and    a.id in
             (select idAppart
              from Propriétaire
              where idPersonne in
                       (select id
                        from Personne
                        where  nom='Grincheux'
                        and prénom='Alice'))
```

En résumé une jointure entre les tables R et S de la forme :

```
select R.*
from R S
where R.a = S.b
```

peut s'écrire de manière équivalente avec une requête imbriquée :

```
select [distinct] *
from R
where R.a in (select S.b from S)
```

ou bien encore sous forme de requête corrélée :

```
select [distinct] *
from R
where exists (select S.b from S where S.b = R.a)
```

Le choix de la forme est matière de goût ou de lisibilité, ces deux critères relevant de considérations essentiellement subjectives.

5.2.3 Requêtes avec négation

Les sous-requêtes sont en revanche irremplaçables pour exprimer des négations. On utilise alors `not in` ou (de manière équivalente) `not exists`. Voici un premier exemple avec la requête : *donner les appartements sans occupant*.

```
select * from Appart
where  id not in (select idAppart from Personne)
```

On obtient comme résultat.

id	no	surface	niveau	idImmeuble
101	34	50	15	1

La négation est aussi un moyen d'exprimer des requêtes courantes comme celle recherchant l'appartement le plus élevé de son immeuble. En SQL, on utilisera typiquement une sous-requête pour prendre le niveau maximal d'un immeuble, et on utilisera cet niveau pour sélectionner un ou plusieurs appartements, le tout avec une requête correlée pour ne comparer que des appartements situés dans le même immeuble.

```
select *
from Appart as a1
where niveau =  (select max(niveau) from Appart as a2
                 where a1.idImmeuble=a2.idImmeuble)
```

id	surface	niveau	idImmeuble	no
101	50	15	1	34
202	250	2	2	2

Il existe en fait beaucoup de manières d'exprimer la même chose. Tout d'abord cette requête peut en fait s'exprimer sans la fonction *max()* avec la négation : si *a* est l'appartement le plus élevé, c'est *qu'il n'existe pas* de niveau plus elevé que *a*. On utilise alors habituellement une requête dite « corrélée » dans laquelle la sous-requête est basée sur une ou plusieurs valeurs issues des tables de la requête principale.

```
select *
from Appart as a1
where not exists  (select * from Appart as a2
```

(suite sur la page suivante)

(suite de la page précédente)

```
                    where a2.niveau > a1.niveau
                    and a1.idImmeuble = a2.idImmeuble)
```

Autre manière d'exprimer la même chose : si le niveau est le plus élevé, tous les autres sont situés à un niveau inférieur. On peut utiliser le mot-clé `all` qui indique que la comparaison est vraie avec *tous* les éléments de l'ensemble constitué par la sous-requête.

```
select *
from Appart as a1
where niveau >= all (select niveau from Appart as a2
                     where a1.idImmeuble=a2.idImmeuble)
```

Dernier exemple de négation : quels sont les personnes qui ne possèdent aucun appartement même partiellement ? Les deux formulations ci-dessous sont équivalentes, l'une s'appuyant sur `not in`, et l'autre sur `not exists`.

```
select *
from Personne
where id not in (select idPersonne from Propriétaire)

select *
from Personne as p1
where not exists (select * from Propriétaire as p2
                  where p1.id=p2.idPersonne)
```

5.3 S3 : Agrégats

Supports complémentaires :
- Diapositives: agrégats
- Vidéo sur les agrégats

Les requêtes d'agrégation en SQL consistent à effectuer des regroupements de nuplets en fonction des valeurs d'une ou plusieurs expressions. Ce regroupement est spécifié par la clause `group by`. On obtient une structure qui n'est pas une table relationnelle puisqu'il s'agit d'un ensemble de groupes de nuplets. On doit ensuite ramener cette structure à une table en appliquant des *fonctions de groupes* qui déterminent des valeurs agrégées calculées pour chaque groupe.

Enfin, il est possible d'exprimer des conditions sur les valeurs agrégées pour ne conserver qu'un ou plusieurs des groupes constitués. Ces conditions portent sur des *groupes* de nuplets et ne peuvent donc être obtenues avec `where`. On utilise alors la clause `having`.

Les agrégats s'effectuent *toujours* sur le résultat d'une requête classique `select - from`. On peut donc les voir comme une extension de SQL consistant à partitionner un résultat en groupes selon certains critères, puis à exprimer des conditions sur ces groupes, et enfin à appliquer des fonctions d'agrégation.

Il existe un groupe par défaut : c'est la table toute entière. Sans même utiliser `group by`, on peut appliquer les fonctions d'agrégation au contenu entier de la table comme le montre l'exemple suivant.

```
select count(*) as nbPersonnes, count(prénom) as nbPrénoms, count(nom) as nbNoms
from Personne
```

Ce qui donne :

nbPersonnes	nbPrénoms	nbNoms
7	6	7

On obtient 7 pour le nombre de nuplets, 6 pour le nombre de prénoms, et 7 pour le nombre de noms. En effet, l'attribut `prénom` est à `null` pour la première personne et n'est en conséquence pas pris en compte par la fonction d'agrégation. Pour compter tous les nuplets, on doit utiliser `count(*)` ou un attribut déclaré comme `not null`. On peut aussi compter le nombre de valeurs distinctes dans un groupe avec `count(distinct <expression>)`.

5.3.1 La clause `group by`

Le rôle du `group by` est de partitionner le résultat d'un bloc `select from where` en fonction d'un critère (un ou plusieurs attributs, ou plus généralement une expression sur des attributs). Pour bien analyser ce qui se passe pendant une requête avec `group by` on peut décomposer l'exécution d'une requête en deux étapes. Prenons l'exemple de celle permettant de vérifier que la somme des quote-part des propriétaires est bien égale à 100 pour tous les appartements.

```
select  idAppart, sum(quotePart) as totalQP
from    Propriétaire
group by idAppart
```

idAppart	totalQP
100	100
101	100
102	100
103	100
104	100
201	100
202	100

Dans une première étape le système va constituer les groupes. On peut les représenter avec un tableau comprenant, pour chaque nuplet, d'une part la (ou les) valeur(s) du (ou des) attribut(s) de partitionnement (ici `idAppart`), d'autre part l'ensemble de nuplets dans lesquelles on trouve cette valeur. Ces nuplets « imbriqués » sont séparés par des points-virgule dans la représentation ci-dessous.

idAppart	Groupe	count
100	(idPersonne=1 quotePart=33 ; idPersonne=5 quotePart=67)	2
101	(idPersonne=1 quotePart=100)	1
102	(idPersonne=5 quotePart=100)	1
103	(idPersonne=2 quotePart=100)	1
104	(idPersonne=2 quotePart=100)	1
201	(idPersonne=5 quotePart=100)	1
202	(idPersonne=1 quotePart=100)	1

Le groupe associé à l'appartement 100 est constitué de deux copropriétaires. Le tableau ci-dessus n'est donc pas une table relationnelle dans laquelle chaque cellule ne peut contenir qu'une seule valeur.

Pour se ramener à une table relationnelle, on transforme durant la deuxième étape chaque groupe de nuplets en une valeur par application d'une fonction d'agrégation. La fonction `count()` compte le nombre de nuplets dans chaque groupe, `max()` donne la valeur maximale d'un attribut parmi l'ensemble des nuplets du groupe, etc. La liste des fonctions d'agrégation est donnée ci-dessous :

- `count(expression)`, Compte le nombre de nuplets pour lesquels `expression` est `not null`.
- `avg(expression)`, Calcule la moyenne de `expression`.
- `min(expression)`, Calcule la valeur minimale de `expression`.
- `max(expression)`, Calcule la valeur maximale de `expression`.
- `sum(expression)`, Calcule la somme de `expression`.
- `std(expression)`, Calcule l'écart-type de `expression`.

Dans la norme SQL l'utilisation de fonctions d'agrégation pour les attributs qui n'apparaissent pas dans le `group by` est *obligatoire*. Une requête comme :

```
select  id, surface, max(niveau) as niveauMax
from    Appart
group by surface
```

sera rejetée parce que le groupe associé à une même surface contient deux appartements différents (et donc deux valeurs différentes pour `id`), et qu'il n'y a pas de raison d'afficher l'un plutôt que l'autre.

5.3.2 Quelques exemples

Note : Vous pouvez exécuter ces requêtes sur le site des TP http://deptfod.cnam.fr/bd/tp.

Calculons la surface totale des appartements, groupés par immeuble. Décomposons : nous avons d'abord besoin du bloc « select - from - where » avec les identifants d'immeubles et les surfaces d'appartement.

```
select idImmeuble, surface
from Appart
```

On ajoute à cette requête la clause `group by` pour grouper par immeuble. On obtient alors (en phase intermédiaire) deux groupes d'appartements, un pour chaque immeuble. Il reste à appliquer une fonction d'agrégation

pour ramener ces groupes à une valeur atomique.

```
select idImmeuble, sum(surface) as totalSurface
from Appart
group by idImmeuble
```

On pourrait aussi appliquer d'autres fonctions d'agrégation :

```
select idImmeuble, min(niveau) as minEtage, max(niveau) as maxEtage,
         sum(surface) as totalSurface
from Appart
group by idImmeuble
```

Revenons un moment à nos voyageurs et à leurs séjours. La requête ci-dessous doit être claire. Exécutez-la sur le site : on constate qu'un voyageur a effectué plusieurs séjours et qu'un logement a reçu plusieurs voyageurs.

```
select v.nom as nomVoyageur, l.nom as nomLogement
from Voyageur as v, Séjour as s, Logement as l
where v.idVoyageur = s.idVoyageur
and l.code = s.codeLogement
```

En ajoutant une clause `group by` on produit des statistiques sur le résultat de cette requête. Par exemple, en groupant sur les voyageurs

```
select v.nom as nomVoyageur, count(*) as 'nbSéjours'
from Voyageur as v, Séjour as s, Logement as l
where v.idVoyageur = s.idVoyageur
and l.code = s.codeLogement
group by v.idVoyageur
```

Ou en groupant sur les logements

```
select l.nom as nomLogement, count(*) as 'nbVoyageurs'
from Voyageur as v, Séjour as s, Logement as l
where v.idVoyageur = s.idVoyageur
and l.code = s.codeLogement
group by l.code
```

On peut aussi regrouper sur plusieurs attributs. Pour obtenir par exemple le nombre de séjours effectués par un voyageur dans un même logement.

```
select l.nom as nomLogement, v.nom as 'nomVoyageur', count(*) as 'nbSéjours'
from Voyageur as v, Séjour as s, Logement as l
where v.idVoyageur = s.idVoyageur
and l.code = s.codeLogement
group by l.code, v.idVoyageur
```

Moralité : à partir d'une requête SQL `select - from -- where`, aussi complexe que nécessaire, vous produisez un résultat (une table). Le `group by` permet d'effectuer des regroupements et des agrégations

(simples) sur ce résultat. Il s'agit vraiment d'un complément au SQL que nous avons étudié en long et en large.

5.3.3 La clause `having`

Finalement, on peut faire porter des conditions sur les groupes, ou plus précisément sur le résultat de fonctions d'agrégation appliquées à des groupes avec la clause `having`. Par exemple, on peut sélectionner les appartements pour lesquels on connaît au moins deux copropriétaires.

```
select  idAppart, count(*) as nbProprios
from    Propriétaire
group by idAppart
having count(*) >= 2
```

On voit que la condition porte ici sur une propriété de l'*ensemble* des nuplets du groupe et pas de chaque nuplet pris individuellement. La clause `having` est donc toujours exprimée sur le résultat de fonctions d'agrégation, par opposition avec la clause `where` qui ne peut exprimer des conditions que sur les nuplets pris un à un.

Important : La requête ci-dessus pourrait s'exprimer en utilisant l'alias pour éviter d'avoir à répeter deux fois le `count(*)` (et pour la rendre plus claire).

```
select  idAppart, count(*) as nbProprios
from    Propriétaire
group by idAppart
having nbProprios >= 2
```

Il n'est malheureusement pas sûr que l'utilisation de l'alias dans le `group by` soit acceptée dans tous les systèmes.

Quelques exemples pour conclure. Toujours sur la base des voyageurs : quels voyageurs ont effectué au moins deux séjours ?

```
select v.nom as nomVoyageur, count(*) as 'nbSéjours'
from Voyageur as v, Séjour as s, Logement as l
where v.idVoyageur = s.idVoyageur
and l.code = s.codeLogement
group by v.idVoyageur
having count(*) > 1
```

Quels logements proposent moins de deux activités.

```
select l.nom
from Logement as l, Activité as a
where l.code = a.codeLogement
group by l.code
having count(*) < 2
```

Voici enfin une requête un peu complexe (sur la base des immeubles) sélectionnant la surface possédée par chaque copropriétaire pour l'immeuble 1. La surface possédée est la somme des surfaces d'appartements possédés par un propriétaire, pondérées par leur quote-part. On regroupe par propriétaire et on trie sur la surface possédée.

```
select prénom nom,
       sum(quotePart * surface / 100) as 'surfacePossédée'
from Personne as p1, Propriétaire as p2, Appart as a
where p1.id=p2.idPersonne
and  a.id=p2.idAppart
and  idImmeuble = 1
group by p1.id
order by sum(quotePart * surface / 100)
```

On obtient le résultat suivant.

nom	surfacePossédée
null	99.5000
Alice	125.0000
Alphonsine	300.5000

5.3.4 Quiz

5.4 S4 : Mises à jour

Supports complémentaires :

Pas de vidéo sur cette partie triviale de SQL.

Les commandes de mise à jour (insertion, destruction, modification) sont considérablement plus simples que les interrogations.

5.4.1 Insertion

L'insertion s'effectue avec la commande `insert`, avec trois variantes. Dans la première on indique la liste des valeurs à insérer sans donner explicitement le nom des attributs. Le système suppose alors qu'il y a autant de valeurs que d'attributs, et que l'ordre des valeurs correspond à celui des attributs dans la table. On peut indiquer `null` pour les valeurs inconnues.

```
insert into Immeuble
values (1 'Koudalou' '3 rue des Martyrs')
```

Si on veut insérer dans une partie seulement des attributs, il faut donner la liste explicitement.

```
insert into Immeuble (id nom adresse)
values (1 'Koudalou' '3 rue des Martyrs')
```

Il est d'ailleurs préférable de toujours donner la liste des attributs. La description d'une table peut changer par ajout d'attribut, et l'ordre `insert` qui marchait un jour ne marchera plus le lendemain.

Enfin avec la troisième forme de `insert` il est possible d'insérer dans une table le résultat d'une requête. Dans ce cas la partie `values` est remplacée par la requête elle-même. Voici un exemple avec une nouvelle table *Barabas* dans laquelle on insère uniquement les informations sur l'immeuble « Barabas ».

```
create table Barabas (id int not null,
                      nom varchar(100) not null,
                      adresse varchar(200),
                      primary key (id)
)

insert into Barabas
select * from Immeuble where nom='Barabas'
```

5.4.2 Destruction

La destruction s'effectue avec la clause `delete` dont la syntaxe est :

```
delete from table
where condition
```

`table` étant bien entendu le nom de la table, et `condition` toute condition ou liste de conditions valide pour une clause `where`. En d'autres termes, si on effectue avant la destruction la requête

```
select * from table
where condition
```

on obtient l'ensemble des nuplets qui seront détruits par `delete`. Procéder de cette manière est un des moyens de s'assurer que l'on va bien détruire ce que l'on souhaite.

5.4.3 Modification

La modification s'effectue avec la clause `update`. La syntaxe est proche de celle du `delete` :

```
update table set A1=v1, A2=v2, ... An=vn
where condition
```

Comme précédemment `table` est la table, les `Ai` sont les attributs les `vi` les nouvelles valeurs, et `condition` est toute condition valide pour la clause `where`.

CHAPITRE 6

Conception d'une base de données

Ce chapitre est consacré la démarche de *conception* d'une base relationnelle. L'objectif de cette conception est de parvenir à un schéma *normalisé* représentant correctement le domaine applicatif à conserver en base de données.

La notion de normalisation a été introduite dans le chapitre *Le modèle relationnel*. Elle s'appuie sur les notions de dépendances fonctionnelles et de clés. On peut, à l'aide de ces notions, caractériser des formes dites « normales ». Peut-on aller plus loin et déterminer comment obtenir une forme normale en partant d'un ensemble global d'attributs liés par des dépendances fonctionnelles ? La première session étudie cette question. Comprendre la normalisation est essentiel pour produire des schémas corrects, viables sur le long terme.

La détermination des clés, des attributs, de leurs dépendances, relève d'une phase de conception. La méthode pratique la plus utilisée est de produire une notation *entité / association*. Elle ne présente pas de difficulté technique mais on constate en pratique qu'elle demande une certaine expérience parce qu'on est confronté à un besoin applicatif pas toujours bien défini, qu'il est difficile de transcrire dans un modèle formel. Les sessions suivantes présentent cette approche et des exemples commentés.

6.1 S1 : La normalisation

Supports complémentaires :

- Diapositives: la normalisation
- Vidéo sur la normalisation

Etant donné un schéma et ses dépendances fonctionnelles, nous savons déterminer s'il est normalisé. Peut-on aller plus loin et produire *automatiquement* un schéma normalisé à partir de l'ensemble des attributs et de leurs contraintes (les DFs) ?

6.1.1 La décomposition d'un schéma

Regardons d'abord le principe avec un exemple illustrant la normalisation d'un schéma relationnel par un processus de décomposition progressif. On veut représenter l'organisation d'un ensemble d'immeubles locatifs en appartements, et décrire les informations relatives aux propriétaires des immeubles et aux occupants de chaque appartement. Voici un premier schéma de relation :

```
Appart(idAppart, surface, idImmeuble, nbEtages, dateConstruction)
```

Voici les dépendances fonctionnelles. La première montre que la clé est `idAppart` : tous les autres attributs en dépendent.

$$idAppart \rightarrow surface, idImmeuble, nbEtages, dateConstruction$$

La seconde représente le fait que l'identifiant de l'immeuble détermine fonctionnellement le nombre d'étages et la date de construction.

$$idImmeuble \rightarrow nbEtages, dateConstruction$$

Cette relation est-elle normalisée ? Non, car la seconde DF montre une dépendance dont la partie gauche n'est pas la clé, `idAppart`. En pratique, une telle relation dupliquerait le nombre d'étages et la date de construction autant de fois qu'il y a d'appartements dans un immeuble.

Une idée naturelle est de prendre les dépendances fonctionnnelles minimales et directes :

$$idAppart \rightarrow surface, idImmeuble$$

et

$$idImmeuble \rightarrow nbEtages, dateConstruction$$

On peut alors créer une table pour chacune. On obtient une décomposition en deux relations :

```
Appart(idAppart, surface, idImmeuble)
Immeuble (idImmeuble, nbEtages, dateConstruction)
```

On n'a pas perdu d'information : connaissant `idAppart`, je connais `idImmeuble`, et connaissant `idImmeuble` je connais les attributs de l'immeuble : je suis donc en mesure de reconstituer l'information initiale. En revanche, j'ai bien éliminé les redondances : les propriétés de l'immeuble ne seront énoncées qu'une seule fois.

Supposons maintenant qu'un immeuble puisse être détenu par plusieurs propriétaires, et considérons la seconde relation suivante, :

```
Proprietaire(idAppart, idPersonne, quotePart)
```

Est-elle normalisée ? Oui car l'unique dépendance fonctionnelle est

$$idAppart, idPersonne \rightarrow quotePart$$

Un peu de réflexion suffit à se convaincre que ni l'appartement, ni le propriétaire ne déterminent à eux seuls la quote-part. Seule l'association des deux permet de donner un sens à cette information, et la clé est donc le couple `(idAppart, idPersonne)`. Maintenant considérons l'ajout du nom et du prénom du propriétaire dans la relation.

```
Propriétaire(idAppart, idPersonne, prénom, nom, quotePart)
```

La dépendance fonctionnelle $idPersonne \rightarrow prnom, nom$ indique que cette relation n'est pas normalisée. En appliquant la même décomposition que précédemment, on obtient le bon schéma :

```
Propriétaire(idAppart, idPersonne, quotePart)
Personne(idPersonne, prénom, nom)
```

Voyons pour finir le cas des occupants d'un appartement, avec la relation suivante.

```
Occupant(idPersonne, nom, prénom, idAppart, surface)
```

On mélange clairement des informations sur les personnes, et d'autres sur les appartements. Plus précisément, la clé est la paire `(idPersonne, idAppart)`, mais on a les dépendances suivantes :

— $idPersonne \rightarrow prnom, nom$
— $idAppart \rightarrow surface$

Un premier réflexe pourrait être de décomposer en deux relations `Personne(idPersonne, prénom, nom)` et `Appart (idAppart, surface)`. Toutes deux sont normalisées, mais on perd alors une information importante, et même essentielle : le fait que telle personne occupe tel appartement. Cette information est représentée par la clé `(idPersonne, idAppart)`. On la préserve en créant une relation `Occupant (idPersonne, idAppart)`. D'où le schéma final :

```
Immeuble (idImmeuble, nbEtages, dateConstruction)
Proprietaire(idAppart, idPersonne, quotePart)
Personne (idPersonne, prénom, nom)
Appart (idAppart, surface, idImmeuble)
Occupant (idPersonne, idAppart)
```

Ce schéma, obtenu par décompositions successives, présente la double propriété

— de ne pas avoir perdu d'information par rapport à la version initiale ;
— de ne contenir que des relations normalisées.

Important : L'absence de perte d'information est une notion qui est survolée ici mais qui est de fait essentielle. Maintenant que nous connaissons SQL, elle est facile à comprendre : l'opération inverse de la décomposition est la *jointure*, effectuée entre la clé primaire d'une table et la clé étrangère référençant cette table. Cette opération reconstitue les données *avant* décomposition, et elle est tellement *naturelle* qu'il existe un opérateur algébrique de ce nom, par exemple :

```
select *
from Appart natural join Immeuble
```

La décomposition d'une table T en plusieurs tables $T_1, T_2, \cdots, T_n$ est sans perte d'information quand on peut reconstituer T avec des jointures $T_1 \bowtie T_2 \bowtie \cdots \bowtie T_n$.

Et voilà. C'est cohérent, simple et élégant.

6.1.2 Algorithme de normalisation

Voici en résumé la procédure de normalisation par décomposition.

Algorithme de normalisation

On part d'un schéma de relation R, et on suppose donné l'ensemble des dépendances fonctionnelles minimales et directes sur R.

On détermine alors les clés de R, et on applique la décomposition :

- Pour chaque DF minimale et directe $X \to A_1, \cdots A_n$ on crée une relation $(X, A_1, \cdots A_n)$ de clé X
- Pour chaque clé C non représentée dans une des relations précédentes, on crée une relation (C) de clé C.

On obtient un schéma de base de données normalisé et sans perte d'information.

Nous disposons donc d'une approche algorithmique pour obtenir un schéma normalisé à partir d'un ensemble initial d'attributs. Cette approche est fondamentalement instructive sur l'objectif à atteindre et la méthode conceptuelle pour y parvenir.

Elle est malheureusement difficilement utilisable telle quelle à cause d'une difficulté rencontrée en pratique : l'absence ou la rareté de dépendances fonctionnelles « naturelles ». Celles présentes dans notre schéma ont été artificiellement créées par ajout d'identifiants pour les immeubles, les occupants et les appartements. Dans la vraie vie, de tels identifiants n'existent pas si l'on n'a pas au préalable déterminé les « entités » présentes dans le schéma : *Immeuble*, *Occupant*, et *Appartement*. En d'autres termes, l'exemple qui précède s'appuie sur une forme de connaissance préalable qui guide à l'avance la décomposition.

La normalisation doit donc être intégrée à une approche plus globale qui « injecte » des dépendances fonctionnelles dans un schéma par identification préalable des entités (les appartements, les immeubles) et des contraintes qu'elles imposent. Le schéma est alors obtenu par application de l'algorithme.

6.1.3 Une approche globale

Reprenons notre table des films pour nous confronter à une situation réaliste. Rappelons les quelques attributs considérés.

```
(titre, année, prénomMes, nomMES, annéeNaiss)
```

La triste réalité est qu'on ne trouve aucune dépendance fonctionnelle dans cet ensemble d'attributs. Le titre d'un film ne détermine rien puisqu'il y a évidemment des films différents avec le même titre, Eventuellement, la paire *(titre, année)* pourrait déterminer de manière univoque un film, mais un peu de réflexion suffit à se

convaincre qu'il est très possible de trouver deux films différents avec le même titre la même année. Et ainsi de suite : le nom du réalisateur ou même la paire *(prénom, nom)* sont des candidats très fragiles pour définir des dépendances fonctionnelles. En fait, on constate qu'il est très rare en pratique de trouver des DFs « naturelles » sur lesquelles on peut solidement s'appuyer pour définir un schéma.

Il nous faut donc une démarche préalable consistant à créer *artificiellement* des DFs parmi les ensembles d'attributs. La connaissance des identifiants d'appartement, d'immeuble et de personne dans notre exemple précédent correspondait à une telle pré-conception : tous les attributs de, respectivement, *Immeuble*, *Appartement* et *Personne*, dépendent fonctionnellement, par construction, de leurs identifiants respectifs, ajoutés au schéma.

Comment trouve-t-on ces identifiants ? Par une démarche consistant à :

— déterminer les « entités » (immeuble, personne, appartement, ou film et réalisateur) pertinents pour l'application ;
— définir une méthode *d'identification* de chaque entité ; en pratique on recourt à la définition d'un *identifiant* artificiel (il n'a aucun rôle descriptif) qui permet d'une part de s'assurer qu'une même « entité » est représentée une seule fois, d'autre part de référencer une entité par son identifiant.
— définir les liens entre les entités.

Voici une illustration informelle de la méthode, que nous reprendrons ensuite de manière plus détailée avec la notation Entité/association.

Commençons par les deux premières étapes. Quelles sont nos entités ? On va décider (il y a dans le processus de conception une part de choix, c'est sa fragilité) que nous avons des entités `Film` et des entités `Réalisateur`. Cela revient à ajouter des identifiants `idFilm` et `idRéalisateur` dans le schéma.

```
(idFilm, titre, année, idRéalisateur, prénom, nom, annéeNaiss)
```

avec les dépendances directes et minimales suivantes :

$$idFilm \rightarrow titre, anne, idRalisateur$$

et

$$idRalisateur \rightarrow prnom, nom, anneNaiss$$

Important : Le choix de l'identifiant est un sujet délicat. On peut arguer en effet que l'identifiant devrait être recherché dans les attributs existants, au lieu d'en créer un artificiellement. Pour des raisons qui tiennent à la rareté/fragilité des DFs « naturelles », la création de l'identifiant artificiel est la seule réellement applicable et satisfaisante dans tous les cas.

À partir de là, il reste à appliquer l'algorithme de normalisation. On obtient une table `Film (idFilm, titre, année, idRéalisateur)` avec une clé primaire et une clé étrangère, et une table `Réalisateur (idRéalisateur, nom, prénom, annéeeNaiss)`.

Important : Il faut veiller à ce que les schémas obtenus soient normalisés. C'est le cas ici puisque les seules DF sont celles issues de l'identifiant.

Voici un exemple pour la table des réalisateurs :

idRéalisateur	titre	année	
101	Scott	Ridley	1943
102	Hitchcock	Alfred	1899
103	Kurosawa	Akira	1910
104	Woo	John	1946
105	Tarantino	Quentin	1963
106	Cameron	James	1954
107	Tarkovski	Andrei	1932

Et pour la table des Films :

idFilm	titre	année	idRéalisateur
1	Alien	1979	101
2	Vertigo	1958	102
3	Psychose	1960	102
4	Kagemusha	1980	103
5	Volte-face	1997	104
6	Pulp Fiction	1995	105
7	Titanic	1997	106
8	Sacrifice	1986	107

Note : La valeur d'un identifiant est locale à une table. On ne peut pas trouver deux fois la même valeur d'identifiant dans une même table, mais rien n'interdit qu'elle soit présente dans deux tables différentes. On aurait donc pu « numéroter » les réalisateurs 1, 2, 3, …, comme pour les films. Ici, nous leur avons donné des identifiants 101, 102, …, pour clarifier les explications.

Cette représentation est correcte. Il n'y a pas de redondance des attributs descriptifs, donc toute mise à jour affecte l'unique occurrence de la donnée à modifier. D'autre part, on peut détruire un film sans affecter les informations sur le réalisateur. La décomposition n'a pas pour contrepartie une perte d'information puisque l'information initiale (autrement dit, *avant* la décomposition en deux tables) peut être reconstituée intégralement. En prenant un film, on obtient l'identifiant de son metteur en scène, et cette identifiant permet de trouver *l'unique* ligne dans la table des réalisateurs qui contient toutes les informations sur ce metteur en scène. Ce processus de reconstruction de l'information, dispersée dans plusieurs tables, peut s'exprimer avec la jointure.

Tout est dit. En maîtrisant la normalisation relationnelle et l'interrogation relationnelle, vous maîtrisez les deux méthodes fondamentales pour la création de bases de données.

6.1.4 Quiz

6.2 S2 : Le modèle Entité-Association

Supports complémentaires :
- Diapositives: le modèle entité / association
- Vidéo sur le modèle entité / association

Cette session présente une méthode complète pour aboutir à un schéma relationnel normalisé. Complète ne veut pas dire infaillible : *aucune* méthode ne produit automatiquement un résultat correct, puisqu'elle repose sur un processus d'analyse et d'identification des entités dont rien ne peut garantir la validité par rapport à un besoin souvent insuffisamment précis.

Le modèle Entité/Association (E/A) propose essentiellement une notation pour soutenir la démarche de conception de schéma présentée précédemment. La notation E/A a pour caractéristiques d'être simple et suffisamment puissante pour modéliser des structures relationnelles. De plus, elle repose sur une représentation graphique qui facilite sa compréhension.

6.2.1 Le schéma de la base *Films*

La présentation qui suit est délibérement axée sur l'utilité du modèle E/A dans le cadre de la conception d'une base de données. Ajoutons qu'il ne s'agit pas directement de *concevoir* un schéma E/A (voir un cours sur les systèmes d'information), mais d'être capable de le comprendre et de l'interpréter. Nous reprenons l'exemple d'une base de données décrivant des films, avec leur metteur en scène et leurs acteurs, ainsi que les cinémas où passent ces films. Nous supposerons également que cette base de données est accessible sur le Web et que des internautes peuvent noter les films qu'ils ont vus.

La méthode permet de distinguer les *entités* qui constituent la base de données, et les *associations* entre ces entités. Un schéma E/A décrit l'application visée, c'est-à-dire une *abstraction* d'un domaine d'étude, pertinente relativement aux objectifs visés. Rappelons qu'une abstraction consiste à choisir certains aspects de la réalité perçue (et donc à éliminer les autres). Cette sélection se fait en fonction de certains *besoins*, qui doivent être précisément définis, et rélève d'une démarche d'analyse qui n'est pas abordée ici.

Par exemple, pour notre base de données *Films*, on n'a pas besoin de stocker dans la base de données l'intégralité des informations relatives à un internaute, ou à un film. Seules comptent celles qui sont importantes pour l'application. Voici le schéma décrivant cete base de données *Films* (figure Fig. 6.1). Sans entrer dans les détails pour l'instant, on distingue
- des *entités*, représentées par des rectangles, ici *Film*, *Artiste*, *Internaute* et *Pays* ;
- des *associations* entre entités représentées par des liens entre ces rectangles. Ici on a représenté par exemple le fait qu'un artiste *joue* dans des films, qu'un internaute *note* des films, etc.

Chaque entité est caractérisée par un ensemble d'attributs, parmi lesquels un ou plusieurs forment l'identifiant unique (en gras). Il est essentiel de dire ce qui caractérise de manière unique une entité, de manière à éviter la redondance d'information. Comme nous l'avons préconisé précédemment, un attribut non-descriptif a été ajouté à chaque entité, indépendamment des attributs « descriptifs ». Nous l'avons appelé **id** pour *Film* et *Artiste*, **code** pour le pays. Le nom de l'attribut-identifiant est peu important, même si la convention **id** est très répandue.

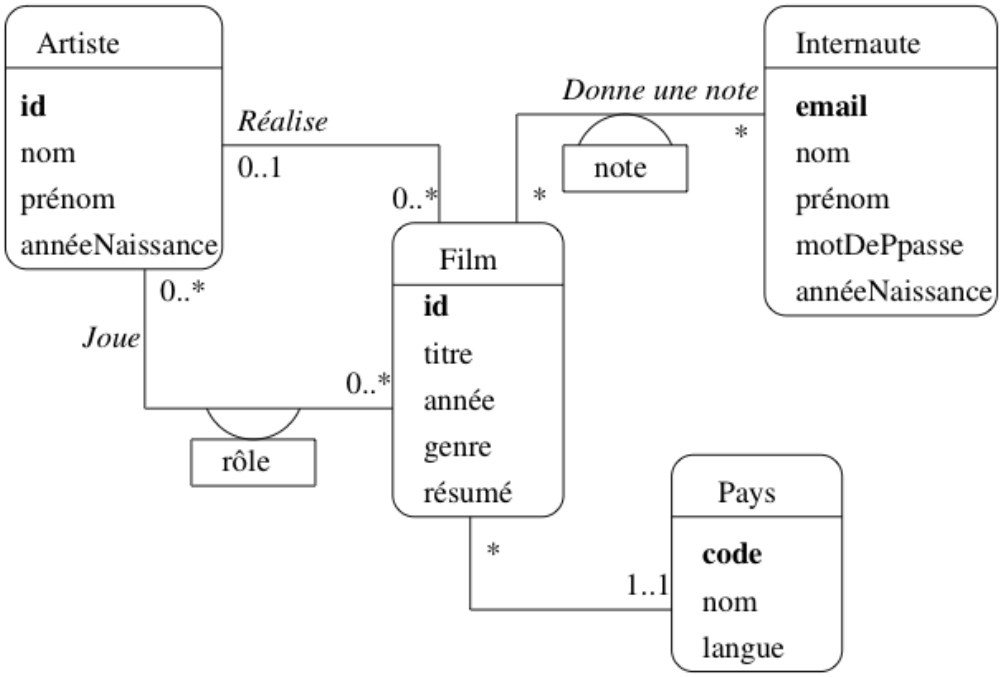

Fig. 6.1 – Le schéma E/A des films

Seule exception : les internautes sont identifiés par un de leurs attributs descriptifs, leur adresse de courrier électronique. Même s'il s'agit en apparence d'un choix raisonnable (unicité de l'email pour identifier une personne), ce cas nous permettra d'illustrer les problèmes qui peuvent quand même se poser.

Les associations sont caractérisées par des *cardinalités*. La notation 0..* sur le lien *Réalise*, du côté de l'entité *Film*, signifie qu'un artiste peut réaliser plusieurs films, ou aucun. La notation 0..1 du côté *Artiste* signifie en revanche qu'un film ne peut être réalisé que par au plus un artiste. En revanche dans l'association *Donne une note*, un internaute peut noter plusieurs films, et un film peut être noté par plusieurs internautes, ce qui justifie l'a présence de 0..* aux deux extrêmités de l'association.

Le choix des cardinalités est *essentiel*. Ce choix est aussi parfois discutable, et constitue donc l'un des aspects les plus délicats de la modélisation. Reprenons l'exemple de l'association *Réalise*. En indiquant qu'un film est réalisé par *un seul* metteur en scène, on s'interdit les – pas si rares – situations où un film est réalisé par plusieurs personnes. Il ne sera donc pas possible de représenter dans la base de données une telle situation. Tout est ici question de choix et de compromis : est-on prêt en l'occurrence à accepter une structure plus complexe (avec 0..* de chaque côté) pour l'association *Réalise*, pour prendre en compte un nombre minime de cas ?

Les cardinalités sont notées par deux chiffres. Le chiffre de droite est la *cardinalité maximale*, qui vaut en général 1 ou *. Le chiffre de gauche est la cardinalité minimale. Par exemple la notation 0..1 entre *Artiste* et *Film* indique qu'on s'autorise à ne pas connaître le metteur en scène d'un film. Attention : cela ne signifie pas que ce metteur en scène n'existe pas. Une base de données, telle qu'elle est décrite par un schéma E/A, ne prétend pas donner une vision exhaustive de la réalité. On ne doit surtout pas chercher à *tout* représenter, mais s'assurer de la prise en compte des besoins de l'application.

La notation 1..1 entre *Film* et *Pays* indique au contraire que l'on doit toujours connaître le pays producteur d'un film. On devra donc interdire le stockage dans la base d'un film sans son pays.

Les cardinalités minimales sont moins importantes que les cardinalités maximales, car elles ont un impact limité sur la structure de la base de données et peuvent plus facilement être remises en cause après coup. Il faut bien être conscient de plus qu'elles ne représentent qu'un choix de conception, souvent discutable. Dans la notation UML que nous présentons ici, il existe des notations abrégées qui donnent des valeurs implicites aux cardinalités minimales :

— La notation * est équivalente à 0..* ;
— la notation 1 est équivalente à 1..1 .

Outre les propriétés déjà évoquées (simplicité, clarté de lecture), évidentes sur ce schéma, on peut noter aussi que la modélisation conceptuelle est totalement indépendante de tout choix d'implantation. Le schéma de la figure *Le schéma E/A des films* ne spécifie aucun système en particulier. Il n'est pas non plus question de type ou de structure de données, d'algorithme, de langage, etc. En principe, il s'agit donc de la partie la plus stable d'une application. Le fait de se débarrasser à ce stade de la plupart des considérations techniques permet de se concentrer sur l'essentiel : que veut-on stocker dans la base ?

Une des principales difficultés dans le maniement des schémas E/A est que la qualité du résultat ne peut s'évaluer que par rapport à une demande qui est difficilement formalisable. Il est donc souvent difficile de mesurer (en fonction de quels critères et quelle métrique ?) l'adéquation du résultat au besoin. Peut-on affirmer par exemple que :

— *toutes* les informations nécessaires sont représentées ?
— qu'un film ne sera *jamais* réalisé par plus d'un artiste ?

Il faut faire des choix, en connaissance de cause, en sachant toutefois qu'il est toujours possible de faire évoluer une base de données, quand cette évolution n'implique pas de restructuration trop importante. Pour reprendre les exemples ci-dessus, il est facile d'ajouter des informations pour décrire un film ou un internaute ; il serait beaucoup plus difficile de modifier la base pour qu'un film passe de un, et un seul, réalisateur, à plusieurs. Quant à changer l'identifiant de la table *Internaute*, c'est une des évolutions les plus complexes à réaliser. Les cardinalités et le choix des clés font vraiment partie des aspects décisifs des choix de conception.

6.2.2 Entités, attributs et identifiants

Il est difficile de donner une définition très précise des entités. Les points essentiels sont résumés par la définition ci-dessous.

Définition : Entités

On désigne par *entité* toute unité d'information *identifiable* et *pertinente* pour l'application.

La notion d'unité d'information correspond au fait qu'une entité ne peut pas se décomposer sans perte de sens. Comme nous l'avons vu précédemment, *l'identité* est primordiale. C'est elle qui permet de distinguer les entités les unes des autres, et donc de dire qu'une information est redondante ou qu'elle ne l'est pas. Il est indispensable de prévoir un moyen technique pour pouvoir effectuer cette distinction entre entités au niveau de la base de données : on parle *d'identifiant* ou (dans un contexte de base de données) de *clé*. Reportez-vous au chapitre *Le modèle relationnel* pour une définition précise de cette notion.

La pertinence est également importante : on ne doit prendre en compte que les informations nécessaires pour satisfaire les besoins. Par exemple :

— le film *Impitoyable* ;
— l'acteur *Clint Eastwood* ;

sont des entités pour la base *Films*.

La première étape d'une conception consiste à identifier les entités utiles. On peut souvent le faire en considérant quelques cas particuliers. La deuxième est de regrouper les entités en ensembles : en général on ne s'intéresse pas à un individu particulier mais à des groupes. Par exemple il est clair que les films et les acteurs constituent des ensembles distincts d'entités. Qu'en est-il de l'ensemble des réalisateurs et de l'ensemble des acteurs ? Doit-on les distinguer ou les assembler ? Il est certainement préférable de les assembler, puisque des acteurs peuvent aussi être réalisateurs.

Attributs

Les entités sont caractérisées par des *attributs* (ou *propriétés*) : le titre (du film), le nom (de l'acteur), sa date de naissance, l'adresse, etc. Le choix des attributs relève de la même démarche d'abstraction qui a dicté la sélection des entités : il n'est pas nécéssaire de donner exhaustivement tous les attributs d'une entité. On ne garde que ceux utiles pour l'application.

Un attribut est désigné par un *nom* et prend sa valeur dans un domaine comme les entiers, les chaînes de caractères, les dates, etc.

Un attribut peut prendre une valeur et une seule. On dit que les attributs sont *atomiques*. Il s'agit d'une restriction importante puisqu'on s'interdit, par exemple, de définir un attribut *téléphones* d'une entité *Personne*, prenant pour valeur *les* numéros de téléphone d'une personne. Cette restricion est l'une des limites du modèle relationnel, qui mène à la multiplication des tables par le mécanisme de normalisation dérit en début de chapitre. Pour notre exemple, il faudrait par exemple définir une table dédiée aux numéros de téléphone et associée aux personnes.

Note : Certaines méthodes admettent l'introduction de constructions plus complexes :
- les *attributs multivalués* sont constitués d'un *ensemble* de valeurs prises dans un même domaine ; une telle construction permet de résoudre le problème des numéros de téléphones multiples ;
- les *attributs composés* sont constitués par agrégation d'autres atributs ; un attribut *adresse* peut par exemple être décrit comme l'agrégation d'un code postal, d'un numéro de rue, d'un nom de rue et d'un nom de ville.

Cette modélisation dans le modèle conceptuel (E/A) doit pouvoir être transposée dans la base de données. Certains systèmes relationnels (PostgreSQL par exemple) autorisent des attributs complexes. Une autre solution est de recourir à d'autres modèles, semi-structurés ou objets.

Nous nous en tiendrons pour l'instant aux attributs atomiques qui, au moins dans le contexte d'une modélisation orientée vers un SGBD relationnel, sont suffisants.

6.2.3 Types d'entités

Il est maintenant possible de décrire un peu plus précisément les entités par leur *type*.

Définition : Type d'entité

Le type d'une entité est composé des éléments suivants :
- son nom ;

— la liste de ses attributs avec, – optionnellement – le domaine où l'attribut prend ses valeurs : les entiers, les chaînes de caractères ;
— l'indication du (ou des) attribut(s) permettant d'identifier l'entité : ils constituent la *clé*.

On dit qu'une entité *e* est une *instance* de son type *E*. Enfin, un ensemble d'entités $\{e_1, e_2, \ldots e_n\}$, instances d'un même type E est une *extension* de E.

Rappelons maintenant la notion de clé, pratiquement identique à celle énoncée pour les schémas relationnels.

Définition : clé

Soit E un type d'entité et A l'ensemble des attributs de E. Une *clé* de E est un sous-ensemble *minimal* de A permettant d'identifier de manière unique une entité parmi n'importe quelle extension de E.

Prenons quelques exemples pour illustrer cette définition. Un internaute est caractérisé par plusieurs attributs : son email, son nom, son prénom, la région où il habite. L'adresse mail constitue une clé naturelle puisqu'on ne trouve pas, en principe, deux internautes ayant la même adresse électronique. En revanche l'identification par le nom seul paraît impossible puisqu'on constituerait facilement un ensemble contenant deux internautes avec le même nom. On pourrait penser à utiliser la paire `(nom,prénom)`, mais il faut utiliser avec modération l'utilisation d'identifiants composés de plusieurs attributs. Quoique possible, elle peut poser des problèmes de performance et complique les manipulations par SQL.

Il est possible d'avoir plusieurs clés candidates pour un même ensemble d'entités. Dans ce cas on en choisit une comme *clé principale* (ou primaire), et les autres comme clés secondaires. Le choix de la clé (primaire) est déterminant pour la qualité du schéma de la base de données. Les caractéristiques d'une bonne clé primaire sont les suivantes :

— elle désigne sans ambiguité une et une seule entité dans toute extension ;
— sa valeur est connue pour toute entité ;
— on ne doit jamais avoir besoin de la modifier ;
— enfin, pour des raisons de performance, sa taille de stockage doit être la plus petite possible.

Il est très difficile de trouver un ensemble d'attributs satisfaisant ces propriétés parmi les attributs descriptifs d'une entité. Considérons l'exemple des films. Le choix du titre pour identifier un film serait incorrect puisqu'on aura affaire un jour ou l'autre à deux films ayant le même titre. Même en combinant le titre avec un autre attribut (par exemple l'année), il est difficile de garantir l'unicité.

Le choix de l'adresse électronique (email) pour un internaute semble respecter ces conditions, du moins la première (unicité). Mais peut-on vraiment garantir que l'email sera connu au moment de la création de l'entité ? De plus, il semble clair que cette adresse peut changer, ce qui va poser de gros problèmes puisque la clé, comme nous le verrons, sert à référencer une entité. Changer l'identifiant de l'entité implique donc de changer *aussi* toutes les références. La conclusion s'impose : ce choix d'identifiant est un mauvais choix, il posera à terme des problèmes pratiques.

Insistons : la seule solution saine et générique consiste à créer un identifiant artificiel, indépendant de tout autre attribut. On peut ainsi ajouter dans le type d'entité *Film* un attribut `id`, corespondant à un numéro séquentiel qui sera incrémenté au fur et à mesure des insertions. Ce choix est de fait le meilleur, dès lors qu'un attribut ne respecte pas les conditions ci-dessus (autrement dit, toujours). Il satisfait ces conditions : on peut toujours lui attribuer une valeur, il ne sera jamais nécessaire de la modifier, et elle a une représentation compacte.

On représente graphiquement un type d'entité comme sur la Fig. 6.2 qui donne l'exemple des types *Internaute* et *Film*. L'attribut (ou les attributs s'il y en a plusieurs) formant la clé sont en gras.

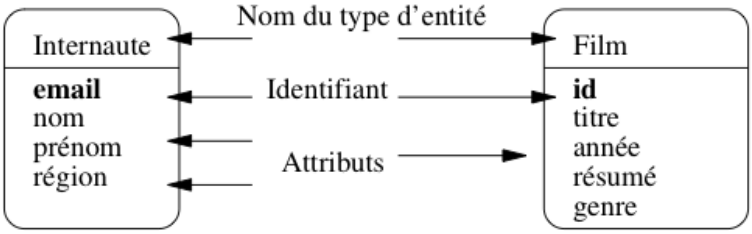

Fig. 6.2 – Représentation des types d'entité

Il est important de bien distinguer *types d'entités* et *entités*. La distinction est la même qu'entre entre *type* et *valeur* dans un langage de programmation, ou *schéma* et *base* dans un SGBD.

6.2.4 Associations binaires

La représentation (et le stockage) d'entités indépendantes les unes des autres est de peu d'utilité. On va maintenant décrire les *relations* (ou *associations*) entre des ensembles d'entités.

Définition : association

Une association binaire entre les ensembles d'entités E_1 et E_2 est un ensemble de couples (e_1, e_2), avec $e_1 \in E_1$ et $e_2 \in E_2$.

C'est la notion classique, ensembliste, de relation. On emploie plutôt le terme d'association pour éviter toute confusion avec le modèle relationnel. Une bonne manière d'interpréter une association entre des ensembles d'entités est de faire un petit graphe où on prend quelques exemples, les plus généraux possibles.

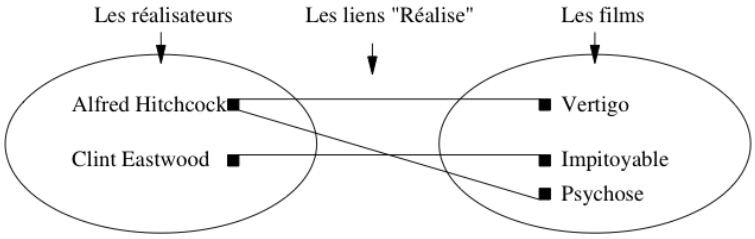

Fig. 6.3 – Association entre deux ensembles

Prenons l'exemple de l'association représentant le fait qu'un réalisateur met en scène des films. Sur le graphe de la Fig. 6.3 on remarque que :

— certains réalisateurs mettent en scène plusieurs films ;
— inversement, un film est mis en scène par au plus un réalisateur.

La recherche des situations les plus générales possibles vise à s'assurer que les deux caractéristiques ci-dessus sont vraies dans tout les cas. Bien entendu on peut trouver x % des cas où un film a plusieurs réalisateurs, mais la question se pose alors : doit-on modifier la structure de notre base, pour x % des cas. Ici, on a décidé

que non. Encore une fois on ne cherche pas à représenter la réalité dans toute sa complexité, mais seulement la partie de cette réalité que l'on veut stocker dans la base de données.

Ces caractéristiques sont essentielles dans la description d'une association entre des ensembles d'entités.

Définition : cardinalités

Soit une association (E_1, E_2) entre deux types d'entités. La cardinalité de l'association pour $E_i, i \in \{1, 2\}$, est une paire $[min, max]$ telle que :

— Le symbole *max* (cardinalité maximale) désigne le nombre *maximal* de fois où une une entité e_i peut intervenir dans l'association.
En général, ce nombre est 1 (au plus une fois) ou n (plusieurs fois, nombre indeterminé), noté par le symbole *.

— Le symbole *min* (cardinalité minimale) désigne le nombre *minimal* de fois où une une entité e_i peut intervenir dans l'association.
En général, ce nombre est 1 (au moins une fois) ou 0.

Les cardinalités maximales sont plus importantes que les cardinalités minimales ou, plus précisément, elles s'avèrent beaucoup plus difficiles à remettre en cause une fois que le schéma de la base est constitué. On décrit donc souvent une association de manière abrégée en omettant les cardinalités minimales. La notation * en UML, est l'abréviation de *0..**, et *1* est l'abréviation de *1..1*. On caractérise également une association de manière concise en donnant les cardinalités maximales aux deux extrêmités, par exemple *1 :** (association de un à plusieurs) ou ** :** (association de plusieurs à plusieurs).

Les cardinalités minimales sont parfois désignées par le terme *contraintes de participation*. La valeur 0 indique qu'une entité peut ne pas participer à l'association, et la valeur 1 qu'elle doit y participer.

Insistons sur le point suivant : *les cardinalités n'expriment pas une vérité absolue, mais des choix de conception*. Elles ne peuvent être déclarés valides que relativement à un besoin. Plus ce besoin sera exprimé précisément, et plus il sera possible d'appécier la qualité du modèle.

Il existe plusieurs manières de noter une association entre types d'entités. Nous utilisons ici la notation de la méthode UML. En France, on utilise aussi couramment – de moins en moins – la notation de la méthode MERISE que nous ne présenterons pas ici.

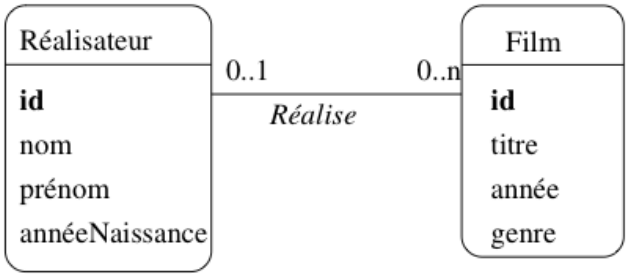

Fig. 6.4 – Représentation de l'association

Dans la notation UML, on indique les cardinalités aux deux extrêmités d'un lien d'association entre deux types d'entités T_A et T_B. Les cardinalités pour T_A sont placées à l'extrémité du lien allant de T_A à T_B et les cardinalités pour T_B sont l'extrémité du lien allant de T_B à T_A.

Pour l'association entre *Réalisateur* et *Film*, cela donne l'association de la Fig. 6.4. Cette association se lit *Un réalisateur réalise zéro, un ou plusieurs films*, mais on pourrait tout aussi bien utiliser la forme passive avec comme intitulé de l'association *Est réalisé par* et une lecture *Un film est réalisé par au plus un réalisateur*. Le seul critère à privilégier dans ce choix des termes est la clarté de la représentation.

Prenons maintenant l'exemple de l'association *(Acteur, Film)* représentant le fait qu'un acteur joue dans un film. Un graphe basé sur quelques exemples est donné dans la Fig. 6.5. On constate tout d'abord qu'un acteur peut jouer dans plusieurs films, et que dans un film on trouve plusieurs acteurs. Mieux : Clint Eastwood, qui apparaissait déjà en tant que metteur en scène, est maintenant également acteur, et dans le même film.

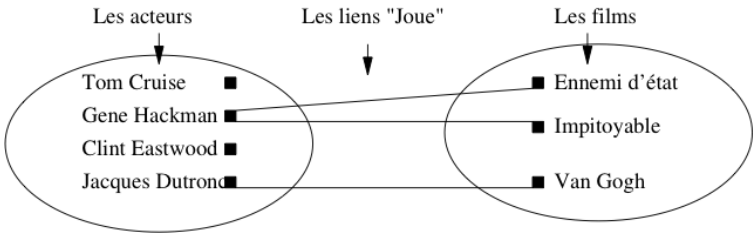

Fig. 6.5 – Association *(Acteur,Film)*

Cette dernière constatation mène à la conclusion qu'il vaut mieux regrouper les acteurs et les réalisateurs dans un même ensemble, désigné par le terme plus général « Artiste ». On obtient le schéma de la Fig. 6.6, avec les deux associations représentant les deux types de lien possible entre un artiste et un film : il peut jouer dans le film, ou le réaliser. Ce « ou » n'est pas exclusif : Eastwood joue dans *Impitoyable*, qu'il a aussi réalisé.

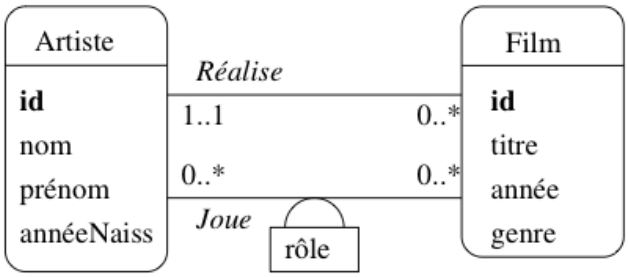

Fig. 6.6 – Association entre *Artiste* et *Film*

Dans le cas d'associations avec des cardinalités multiples de chaque côté, on peut avoir des attributs qui ne peuvent être affectés qu'à l'association elle-même. Par exemple l'association *Joue* a pour attribut le rôle tenu par l'acteur dans le film (Fig. 6.6).

Rappelons qu'un attribut ne peut prendre qu'une et une seule valeur. Clairement, on ne peut associer `rôle` ni à *Acteur* puisqu'il a autant de valeurs possibles qu'il y a de films dans lesquels cet acteur a joué, ni à *Film*, la réciproque étant vraie également. Seules les associations ayant des cardinalités multiples de chaque côté peuvent porter des attributs.

Quelle est la clé d'une association ? Si l'on s'en tient à la définition, une association est un *ensemble* de couples, et il ne peut donc y avoir deux fois le même couple (parce qu'on ne trouve pas deux fois le même élément dans un ensemble). On a donc :

Définition : Clé d'une association

La clé d'une association (binaire) entre un type d'entité E_1 et un type d'entité E_2 est la paire constituée de la clé c_1 de E_1 et de la clé c_2 de E_2.

Cette contrainte est parfois trop contraignante car on souhaite autoriser deux entités à être liées plus d'une fois dans une association.

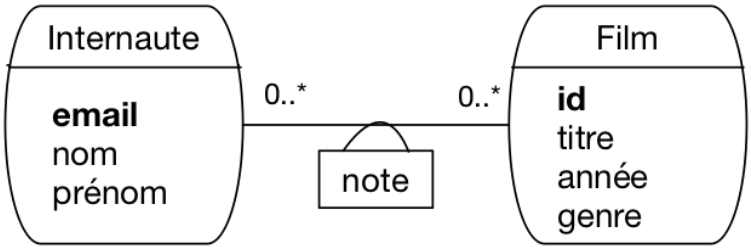

Fig. 6.7 – Association entre *Internaute* et *Film*

Prenons le cas de l'association entre *Internaute* et *Film* (Fig. 6.7). Elle est identifiée par la paire `(idFilm, email)`. Imaginons par exemple qu'un internaute soit amené à noter à plusieurs reprises un film, et que l'on souhaite conserver l'historique de ces notations successives. Avec une association binaire entre *Internaute* et *Film*, c'est impossible : on ne peut définir qu'un seul lien entre un film donné et un internaute donné.

Le problème est qu'il n'existe pas de moyen pour distinguer des liens multiples entre deux mêmes entités. Dans ce type de situation, on peut considérer que l'association est porteuse de trop sens et qu'il vaut mieux la *réifier* sous la forme d'une entité.

Note : La réification consiste à transformer en un objet réel et autonome une idée ou un concept. Ici, le concept d'association entre deux entités est réifié en lui donnant le statut d'entité, avec pour principale conséquence l'attribution d'un identifiant autonome. Cela signifie qu'une telle entité pourrait, en principe, exister indépendamment des entités de l'association initiale. Quelques précautions s'imposent, mais en pratique, c'est une option tout à fait valable.

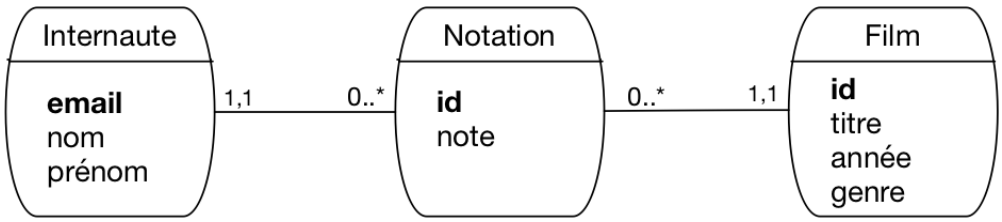

Fig. 6.8 – réification de l'association entre *Internaute* et *Film*

La Fig. 6.8 montre le modèle obtenu en transformant l'association *Note* en entité *Notation*. Notez soigneusement les caractéristiques de cette transformation :

— chaque note a maintenant un identifiant propre, `id`
— une entité *Note* est liée par une association « plusieurs à un » respectivement à un film et à un internaute ; cela reflète l'origine de cette entité, représentant un lien entre une paire d'entités (Film, Internaute).

— Nous avons mis une contrainte de participation forte (1..1) pour ces associations, afin de conserver l'idée qu'une note ne saurait exister sans que l'on connaisse le film d'une part, l'internaute d'autre part.

Le choix de réifier ou non une association plusieurs-plusieurs relève du jugement. Dès qu'une association porte beaucoup d'information et des contraintes un peu complexes, il est sans doute préférable de la transformer en entité.

Note : Certains outils de conception ne permettent que les associations un-à-plusieurs, ce qui impose de fait de réifier les associations plusieurs-plusieurs.

6.2.5 Quiz

6.3 S3 : Concepts avancés

Supports complémentaires :

— Diapositives: concepts avancés de modélisation
— Vidéo sur les concepts avancés de modélisation / association

---

Cette session présente quelques extensions courantes aux principes de base du modèle entité-association.

6.3.1 Entités faibles

Jusqu'à présent nous avons considéré le cas d'entités *indépendantes* les unes des autres. Chaque entité, disposant de son propre identifiant, pouvait être considérée isolément. Il existe des cas où une entité ne peut exister qu'en étroite association avec une autre, et est identifiée relativement à cette autre entité. On parle alors *d'entité faible*.

Prenons l'exemple d'un cinéma, et de ses salles. On peut considérer chaque salle comme une entité, dotée d'attributs comme la capacité, l'équipement en son Dolby, ou autre. Il est diffcilement imaginable de représenter une salle sans qu'elle soit rattachée à son cinéma. C'est en effet au niveau du cinéma que l'on va trouver quelques informations générales comme l'adresse de la salle.

Il est possible de représenter le lien en un cinéma et ses salles par une association classique, comme le montre la Fig. 6.9.a. La cardinalité 1..1 force la participation d'une salle à un lien d'association avec un et un seul cinéma. Cette représentation est correcte, mais présente un (léger) inconvénient : on doit créer un identifiant artificiel `id` pour le type d'entité *Salle*, et numéroter toutes les salles, *indépendamment du cinéma auquel elles sont rattachées*.

On peut considérer qu'il est plus naturel de numéroter les salles par un numéro interne à chaque cinéma. La clé d'identification d'une salle est alors constituée de deux parties :

— la clé de *Cinéma*, qui indique dans quel cinéma se trouve la salle ;
— le numéro de la salle au sein du cinéma.

En d'autres termes, l'entité *Salle* ne dispose pas d'une identification absolue, mais d'une identification *relative* à une autre entité. Bien entendu cela force la salle a toujours être associée à un et un seul cinéma.

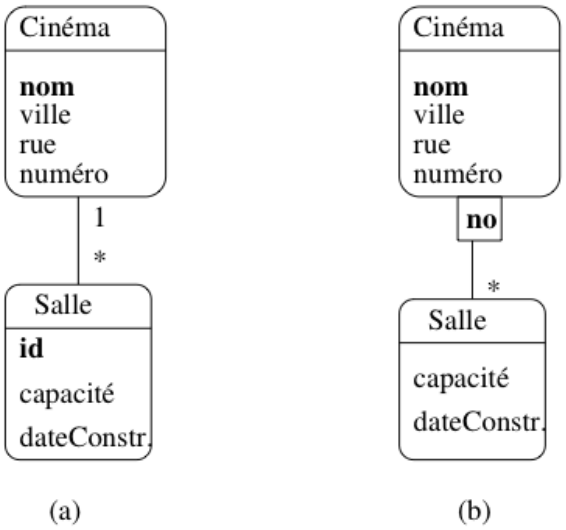

Fig. 6.9 – Modélisations possibles du lien Cinéma-Salle

La représentation graphique des entités faibles avec UML est illustrée dans la Fig. 6.9.b. La salle est associée au cinéma avec une association qualifiée par l'attribut `no` qui sert de discriminant pour distinguer les salles au sein d'un même cinéma. Noter que la cardinalité du côté *Cinéma* est implicitement 1..1.

L'introduction d'entités faibles est un subtilité qui permet de capturer une caractéristique intéressante du modèle. Elle n'est pas une nécessité absolue puisqu'on peut très bien utiliser une association classique. La principale différence est que, dans le cas d'une entité faible, on obtient une identification composée qui peut être plus pratique à gérer, et peut également rendre plus faciles certaines requêtes. On touche ici à la liberté de choix qui est laissée, sur bien des aspects, à un « modeleur » de base de données, et qui nécessite de s'appuyer sur une expérience robuste pour apprécier les conséquences de telle ou telle décision.

La présence d'un type d'entité faible B associé à un type d'entité A implique également des contraintes fortes sur les créations, modifications et destructions des instances de A car on doit toujours s'assurer que la contrainte est valide. Concrètement, en prenant l'exemple de *Salle* et de *Cinéma*, on doit mettre en place les mécanismes suivants :

— quand on insère une salle dans la base, on doit toujours l'associer à un cinéma ;
— quand un cinéma est détruit, on doit aussi détruire toutes ses salles ;
— quand on modifie la clé d'un cinéma, il faut répercuter la modification sur toutes ses salles (mais il est préférable de ne jamais avoir à modifier une clé).

Réfléchissez bien à ces mécanismes pour apprécier le sucroît de contraintes apporté par des variantes des associations. Parmi les impacts qui en découlent, et pour respecter les règles de destruction/création énoncées, on doit mettre en place une stratégie. Nous verrons que les SGBD relationnels nous permettent de spécifier de telles stratégies.

6.3.2 Associations généralisées

On peut envisager des associations entre plus de deux entités, mais elles sont plus difficiles à comprendre, et surtout la signification des cardinalités devient beaucoup plus ambigue. Prenonsl'exemple d'une association permettant de représenter la projection de certains films dans des salles. Une association plusieurs-à-plusieurs entre *Film* et *Salle* semble à priori faire l'affaire, mais dans ce cas l'identifiant de chaque lien est la paire constituée `(idFilm, idSalle)`. Cela interdit de projeter plusieurs fois le même film dans la même salle, ce qui pour le coup est inacceptable.

Il faut donc introduire une information supplémentaire, par exemple l'horaire, et définir association ternaire entre les types d'entités *Film*, *Salle* et *Horaire*. On obtient la Fig. 6.10.

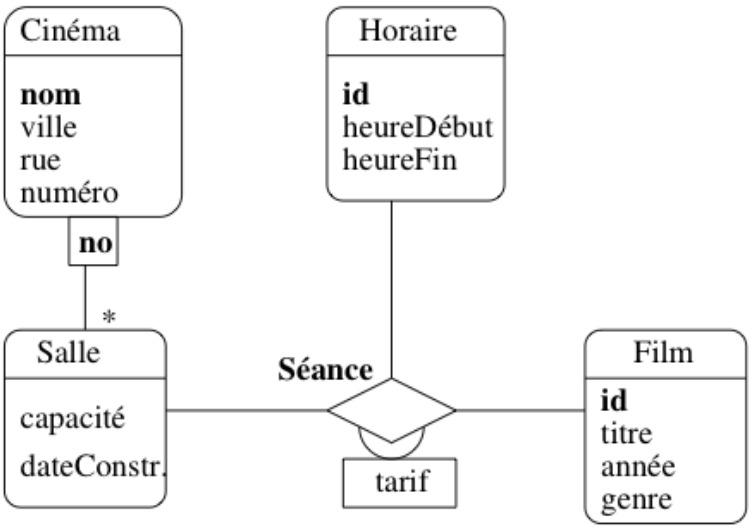

Fig. 6.10 – Association ternaire représentant les séances

On tombe alors dans plusieurs complications. Tout d'abord les cardinalités sont, implicitement, 0..*. Il n'est pas possible de dire qu'une entité ne participe qu'une fois à l'association. Il est vrai que, d'une part la situation se présente rarement, d'autre part cette limitation est due à la notation UML qui place les cardinalités à l'extrémité opposée d'une entité.

Plus problématique en revanche est la détermination de la clé. Qu'est-ce qui identifie un lien entre trois entités ? En principe, la clé est le triplet constitué des clés respectives de la salle, du film et de l'horaire constituant le lien. On aurait donc le n-uplet *[nomCinéma, noSalle, idFilm, idHoraire]*. Une telle clé ne permet pas d'imposer certaines contraintes comme, par exemple, le fait que dans une salle, pour un horaire donné, il n'y a qu'un seul film.

Ajouter une telle contrainte, c'est signifier que la clé de l'association est en fait constitué de *[nomCinéma, noSalle, idHoraire]*. C'est donc un sous-ensemble de la concaténation des clés, ce qui semble rompre avec la définition donnée précédemment. Inutile de développer plus : les associations de degré supérieur à deux sont difficiles à manipuler et à interpréter. Il est *toujours* possible d'utiliser le mécanisme de réification déjà énoncé et de remplacer cette association par un type d'entité. Pour cela on suit la règle suivante :

Règle de réification

Soit A une association entre les types d'entité $\{E_1, E_2, \ldots, E_n\}$. La transformation de A en type d'entité

E_A s'effectue en deux étapes :

— On attribue un identifiant autonome à E_A.
— On crée une association A_i de type "(0..n) :(1..1)" entre E_A et chacun des E_i.

Chaque instance de E_A doit impérativement être associée à une instance (et une seule) de chacun des E_i. Cette contrainte de participation (le « 1..1 ») est héritée du fait que l'association initiale ne pouvait exister que comme lien créé entre les entités participantes.

L'association précédente peut être transformée en un type d'entité *Séance*. On lui attribue un identifiant `idSéance`, et des associations "(0..n) :(1..1)" avec *Film*, *Horaire* et *Salle*. Pour chaque séance, on doit impérativement connaître le fillm, la salle et l'horaire. On obtient le schéma de la Fig. 6.11.

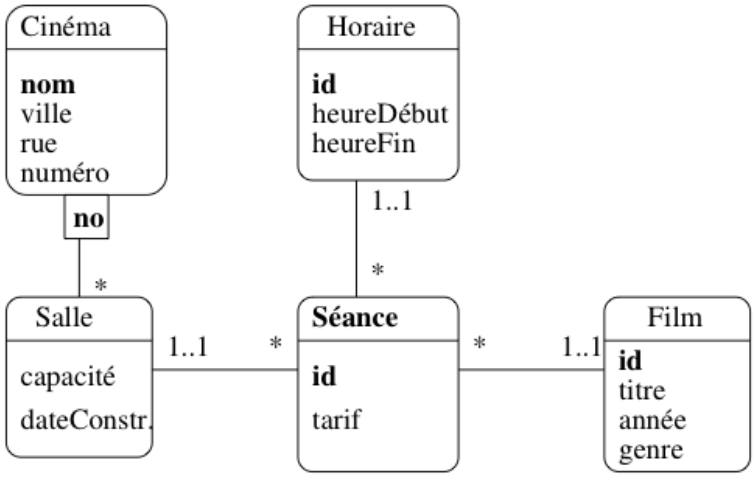

Fig. 6.11 – L'association *Séance* transformée en entité

On peut ensuite ajouter des contraintes supplémentaires, indépendantes de la clé, pour dire par exemple qu'il ne peut y avoir qu'un film dans une salle pour un horaire. Nous étudierons ces contraintes dans le prochaine chapitre.

6.3.3 Spécialisation

La modélisation UML est basée sur le modèle orienté-objet dont l'un des principaux concepts est *la spécialisation*. On est ici au point le plus divergent des représentations relationnelle et orienté-objet, puisque la spécialisation n'existe pas dans la première, alors qu'il est au cœur des modélisations avancées dans la seconde.

Il n'est pas très fréquent d'avoir à gérer une situation impliquant de la spé&cialisation dans une base de données. Un cas sans doute plus courant est celui où on effectue la modélisation objet d'une application dont on souhaite rendre les données persistantes.

Note : Cette partie présente des notions assez avancées qui sont introduites pour des raisons de complétude mais peuvent sans trop de dommage être ignorées dans un premier temps.

Voici quelques brèves indications, en prenant comme exemple illustratif le cas très simple d'un raffinement de notre modèle de données. La notion plus générale de *vidéo* est introduite, et un film devient un cas particulier de vidéo. Un autre cas particulier est le *reportage*, ce qui donne donc le modèle de la Fig. 6.12. Au niveau de la super-classe, on trouve le titre et l'année. Un film se distingue par l'association à des acteurs et un metteur en scène ; un reportage en revanche a un lieu de tournage et une date.

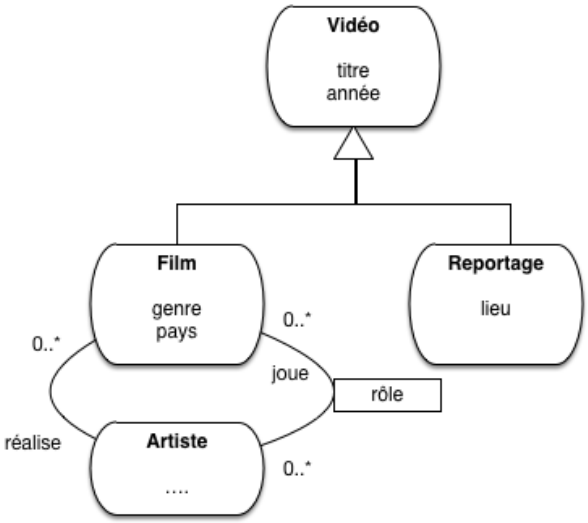

Fig. 6.12 – Notre exemple d'héritage

Le type d'entité *Vidéo* factorise les propriétés commune à toutes les vidéos, quel que soit leur type particulier (film, ou reportage, ou dessin animé, ou n'importe quoi d'autre). Au niveau de chaque sous-type, on trouve les propriétés particulières, par exemple l'association avec les acteurs pour un film (qui n'a pas de sens pour un reportage ou un dessin animé).

La particularité de cette modélisation est qu'une entité (par exemple un film) voit sa représentation éclatée selon deux types d'entité, cas que nous n'avons pas encore rencontré.

6.3.4 Bilan

Le modèle E/A est l'outil universellement utilisé pour modéliser une base de données. Il présente malheureusement plusieurs limitations, qui découlent du fait que beaucoup de choix de conceptions plus ou moins équivalents peuvent découler d'une même spécification, et que la spécification elle-même est dans la plupart du cas informelle et sujette à interprétation.

Un autre inconvénient du modèle E/A reste sa pauvreté : il est difficile d'exprimer des contraintes d'intégrité, des structures complexes. Beaucoup d'extensions ont été proposées, mais la conception de schéma reste en partie matière de bon sens et d'expérience. On essaie en général :

— de se ramener à des associations entre 2 entités : au-delà, on a probablement intérêt a transformer l'association en entité ;
— d'éviter toute redondance : une information doit se trouver en un seul endroit ;
— enfin – et surtout – de privilégier la simplicité et la lisibilité, notamment en ne représentant que ce qui est strictement nécessaire.

La mise au point d'un modèle engage fortement la suite d'un projet de développement de base de données. Elle doit s'appuyer sur des personnes expérimentées, sur l'écoute des prescripteurs, et sur un processus par itération qui identifie les ambiguités et cherche à les résoudre en précisant le besoin correspondant.

Dans le cadre des bases de données, le modèle E/A est utilisé dans la phase de conception. Il permet de spécifier tout ce qui est nécessaire pour construire un schéma de base de données normalisé, comme expliqué dans la prochaine session.

6.3.5 Quiz

6.4 S4 : Du schéma E/A au schéma relationnel

Supports complémentaires :

— Diapositives: de la modélisation EA au schéma relationnel
— Vidéo sur le passage de la modélisation EA au schéma relationnel / association

---

La création d'un schéma de base de données est simple une fois que le schéma entité/association est finalisé. Il suffit d'appliquer l'algorithme de normalisation vu en début de chapitre. Cette session est essentiellement une illustration de cet algorithme appliqué à la base de films, agrémentée d'une discussion sur quelques cas particuliers.

6.4.1 Application de la normalisation

Pour rappel, voici le schéma E/A de la base des films (Fig. 6.13), discuté précédemment.

Ce schéma donne toutes les informations nécessaires pour appliquer l'algorithme de normalisation vu en début de chapitre.

— Chaque entité définit une dépendance fonctionnelle minimale et directe de l'identifiant vers l'ensemble des attributs. On a par exemple pour l'entité `Film` :

$$idFilm \rightarrow titre, anne, genre, rsum$$

— Chaque association plusieurs-à-un correspond à une dépendance fonctionnelle minimale et directe entre l'identifiant de la première entité et l'identifiant de la seconde. Par exemple, l'association « Réalise » entre `Film` et `Artiste`` définit la DF :

$$idFilm \rightarrow idArtiste$$

On peut donc ajouter `idArtiste` à la liste des attributs dépendants de `idFilm`.

— Enfin chaque association (binaire) plusieurs-à-plusieurs correspond à une dépendance fonctionnelle minimale et directe entre l'identifiant de l'association (qui est la paire des identifiants provenant des entités liées par l'association) et les attributs propres à l'association.
Par exemple, l'association `Joue` définit la DF

$$(idFilm, idArtiste) \rightarrow rle$$

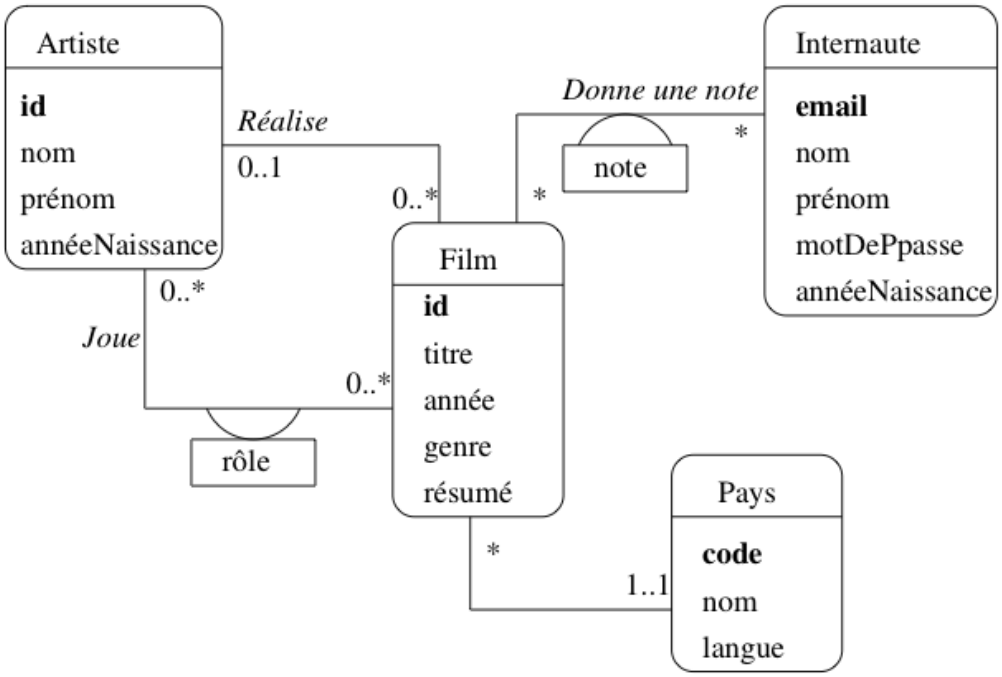

Fig. 6.13 – Le schéma E/A des films

Important : Si une association plusieurs-à-plusieurs n'a pas d'attribut propre, il faut quand même penser à créer une relation avec la clé de l'association (autrement dit la paire des identifiants d'entité) pour conserver l'information sur les liens entre ces entités.

Exemple : la Fig. 6.14 montre une association plusieurs-plusieurs entre `Film` et `Internaute`, sans attribut propre. Il ne faut pas oublier dans ce cas de créer une table `Vu(idFilm, email)` constituée simplement de la clé. Elle représente le lien entre un film et un internaute.

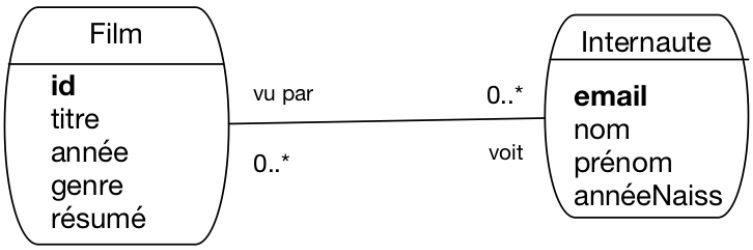

Fig. 6.14 – L'association « Un internaute a vu un film »

Et c'est tout. En appliquant l'algorithme de normalisation à ces DF, on obtient le schéma normalisé suivant :

- Film (**idFilm**, titre, année, genre, résumé, *idArtiste*, *codePays*)
- Artiste (**idArtiste**, nom, prénom, annéeNaissance)
- Pays (**code**, nom, langue)
- Role (**idFilm, idActeur**, nomRôle)
- Notation (**email, idFilm**, note)

Les clés primaires sont en gras : ce sont les identifiants des entités ou des associations plusieurs-à-plusieurs.

Les attributs qui proviennent d'une DF définie par une association, comme par exemple $idFilm \rightarrow idArtiste$, sont en italiques pour indiquer leur statut particulier : ils servent de référence à une entité representée par un autre nuplet. Ces attributs sont les *clé étrangères* de notre schéma.

Comment nommer la clé étrangère ? Ici nous avons adopté une convention simple en concaténant `id` et le nom de la table référencée. On peut souvent faire mieux. Par exemple, dans le schéma de la table *Film*, le rôle précis tenu par l'artiste référencé dans l'association n'est pas induit par le nom `idArtiste`. L'artiste dans *Film* a un rôle de metteur en scène, mais il pourrait tout aussi bien s'agir du décorateur ou de l'accessoiriste : rien dans le nom de l'attribut ne le précise

On peut donner un nom plus explicite à l'attribut. Il n'est pas du tout obligatoire en fait que les attributs constituant une clé étrangère aient le même nom que ceux de le clé primaire auxquels ils se réfèrent. Voici le schéma de la table *Film*, dans lequel la clé étrangère pour le metteur en scène est nommée `idRéalisateur`.

— Film (**idFilm**, titre, année, genre, résumé, *idRéalisateur*, *codePays*)

Le schéma E/A nous fournit donc une sorte de résumé des spécifications suffisantes pour un schéma normalisé. Il n'y a pas grand chose de plus à savoir. Ce qui suit donne une illustration des caractéristiques de la base obtenue, et quelques détails secondaires mais pratiques.

6.4.2 Illustration avec la base des films

Les tables ci-dessous montrent un exemple de la représentation des associations entre *Film* et *Artiste* d'une part, *Film* et *Pays* d'autre part (on a omis le résumé du film).

id	nom	prénom	année
101	Scott	Ridley	1943
102	Hitchcock	Alfred	1899
103	Kurosawa	Akira	1910
104	Woo	John	1946
105	Tarantino	Quentin	1963
106	Cameron	James	1954
107	Tarkovski	Andrei	1932

Noter que l'on ne peut avoir qu'un artiste dont l'`id` est 102 dans la table *Artiste*, puisque l'attribut `idArtiste` ne peut prendre qu'une valeur. Cela correspond à la contrainte, identifiée pendant la conception et modélisée dans le schéma E/A de la Fig. 6.13, qu'un film n'a qu'un seul réalisateur.

En revanche rien n'empêche cet artiste 102 de figurer plusieurs fois dans la colonne `idRéalisateur` de la table *Film* puisqu'il n'y a aucune contrainte d'unicité sur cet attribut. On a donc bien l'équivalent de l'association un à plusieurs élaborée dans le schéma E/A.

Et voici la table des films. Remarquez que chaque valeur de la colonne `idRéalisateur` est l'identifiant d'un artiste.

id	titre	année	genre	idRéali-sateur	code-Pays
1	Alien	1979	Science-Fiction	101	USA
2	Vertigo	1958	Suspense	102	USA
3	Psychose	1960	Suspense	102	USA
4	Kagemusha	1980	Drame	103	JP
5	Volte-face	1997	Policier	104	USA
6	Pulp Fiction	1995	Policier	105	USA
7	Titanic	1997	Drame	106	USA
8	Sacrifice	1986	Drame	107	FR

Note : Les valeurs des clés primaires et étrangères sont complètement indépendantes l'une de l'autre. Nous avons identifié les films en partant de 1 et les artistes en partant de 101 pour des raisons de clarté, mais en pratique rien n'empêche de trouver une ligne comme :

(63, Gravity, 2014, SF, 63, USA)

Il n'y a pas d'ambiguité : le premier "63" est l'identifiant du film, le second est l'identifiant du réalisateur.

Et voici, pour compléter, la table des pays.

code	nom	langue
USA	Etats Unis	anglais
FR	France	français
JP	Japon	japonais

Pour bien comprendre le mécanisme de representation des entités et associations grâce aux clés primaires et étrangères, examinons les tables suivantes montrant un exemple de représentation de *Rôle*. On peut constater le mécanisme de référence unique obtenu grâce aux clés des tables. Chaque rôle correspond à un unique acteur et à un unique film. De plus on ne peut pas trouver deux fois la même paire `(idFilm, idActeur)` dans cette table (c'est un choix de conception qui découle du schéma E/A sur lequel nous nous basons). En revanche un même acteur peut figurer plusieurs fois (mais pas associé au même film), ainsi qu'un même film (mais pas associé au même acteur).

Voici tout d'abord la table des films.

id	titre	année	genre	idRéali-sateur	code-Pays
20	Impitoyable	1992	Western	130	USA
21	Ennemi d'état	1998	Action	132	USA

Puis la table des artistes.

id	nom	prénom	année
130	Eastwood	Clint	1930
131	Hackman	Gene	1930
132	Scott	Tony	1930
133	Smith	Will	1968

En voici la table des rôles, qui consiste ensentiellement en identifiants établissant des liens avec les deux tables précédentes. À vous de les décrypter pour comprendre comment toute l'information est représentée, et conforme aux choix de conception issus du schéma E/A. Que peut-on dire de l'artiste 130 par exemple ? Peut-on savoir dans quels films joue Gene Hackman ? Qui a mis en scène *Impitoyable* ?

idFilm	idArtiste	nomRôle
20	130	William Munny
20	131	Little Bill
21	131	Bril
21	133	Robert Dean

On peut donc remarquer que chaque partie de la clé de la table *Rôle* est elle-même une clé étrangère qui fait référence à une ligne dans une autre table :

— l'attribut `idFilm` fait référence à une ligne de la table *Film* (un film) ;
— l'attribut `idActeur` fait référence à une ligne de la table *Artiste* (un acteur) ;

Le même principe de référencement et d'identification des tables s'applique à la table *Notation*. Il faut bien noter que, par choix de conception, on a interdit qu'un internaute puisse noter plusieurs fois le même film, de même qu'un acteur ne peut pas jouer plusieurs fois dans un même film. Ces contraintes ne constituent pas des limitations, mais des décisions prises au moment de la conception sur ce qui est autorisé, et sur ce qui ne l'est pas.

6.4.3 Associations avec type d'entité faible

Une entité faible est toujours identifiée par rapport à une autre entité. C'est le cas par exemple de l'association entre *Cinéma* et *Salle* (voir session précédente). Cette association est de type « un à plusieurs » car l'entité faible (une salle) est liée à une seule autre entité (un cinéma) alors que, en revanche, un cinéma peut être lié à plusieurs salles.

Le passage à un schéma relationnel est donc identique à celui d'une association 1-*n* classique. On utilise un mécanisme de clé étrangère pour référencer l'entité forte dans l'entité faible. La seule nuance est que la clé étrangère est une partie de l'identifiant de l'entité faible.

Regardons notre exemple pour bien comprendre. Voici le schéma obtenu pour représenter l'association entre les types d'entité *Cinéma* et *Salle*.

— Cinéma (**id**, nom, numéro, rue, ville)
— Salle (**idCinéma, no**, capacité)

On note que l'identifiant d'une salle est constitué de l'identifiant du cinéma et d'un numéro complémentaire permettant de distinguer les salles au sein d'un même cinéma. Mais l'identifiant du cinéma dans *Salle* est *aussi* une clé étrangère référençant une ligne de la table *Cinéma*. En d'autres termes, la clé étrangère est une partie de la clé primaire.

Cette modélisation simplifie l'attribution de l'identifiant à une nouvelle entité *Salle* puisqu'il suffit de reprendre l'identifiant du composé (le cinéma) et de numéroter les composants (les salles) relativement au composé. Il ne s'agit pas d'une différence vraiment fondamentale avec les associations *1-n* mais elle peut clarifier le schéma.

6.4.4 Spécialisation

Note : La spécialisation est une notion avancée dont la représentation en relationnel n'est pas immédiate. Vous pouvez omettre d'étudier cette partie dans un premier temps.

Pour obtenir un schéma relationnel représentant la spécialisation, il faut trouver un contournement. Voici les trois solutions possibles pour notre spécialisation Vidéo-Film-Reportage. Aucune n'est idéale et vous trouverez toujours quelqu'un pour argumenter en faveur de l'une ou l'autre. Le mieux est de vous faire votre propre opinion (je vous donne la mienne un peu plus loin).

— *Une table pour chaque classe*. C'est la solution la plus directe, menant pour notre exemple à créer des tables *Vidéo*, *Film* et *Reportage*. Remarque très importante : on doit *dupliquer* dans la table d'une sous-classe les attributs persistants de la super-classe. Le titre et l'année doivent donc être dupliqués dans, respectivement, *Film* et *Reportage*. Cela donne des tables indépendantes, chaque objet étant complètement représenté par une seule ligne.
Remarque annexe : si on considère que *Vidéo* est une classe abstraite qui ne peut être instanciée directement, on ne crée pas de table *Vidéo*.

— *Une seule table pour toute la hiérarchie d'héritage*. On créerait donc une table *Vidéo*, et on y placerait tous les attributs persistants de *toutes* les sous-classes. La table *Vidéo* aurait donc un attribut *id_realisateur* (venant de *Film*), et un attribut *lieu* (venant de *Reportage*).
Les instances de *Vidéo*, *Film* et *Reportage* sont dans ce cas toutes stockées dans la même table *Vidéo*, ce qui nécessite l'ajout d'un attribut, dit *discriminateur*, pour savoir à quelle classe précise correspondent les données stockées dans une ligne de la table. L'inconvénient évident, surtout en cas de hiérarchie complexe, est d'obtenir une table fourre-tout contenant des données difficilement compréhensibles.

— Enfin, la troisième solution est un mixte des deux précédentes, consistant à créer une table par classe (donc, trois tables pour notre exemple), tout en gardant la spécialisation propre au modèle d'héritage : chaque table ne contient que les attributs venant de la classe à laquelle elle correspond, et une *jointure* permet de reconstituer l'information complète.
Par exemple : un film serait représenté partiellement (pour le titre et l'année) dans la table *Vidéo*, et partiellement (pour les données qui lui sont spécifiques, comme *id_realisateur*) dans la table *Film*.

Aucune solution n'est totalement satisfaisante, pour les raisons indiquées ci-dessus. Voici une petite discussion donnant mon avis personnel.

La duplication introduite par la première solution semble source de problèmes à terme, et je ne la recommande vraiment pas. Tout changement dans la super-classe devrait être répliqué dans toutes les sous-classes, ce qui donne un schéma douteux et peu contrôlable.

Tout placer dans une même table se défend, et présente l'avantage de meilleures performances puisqu'il n'y a pas de jointure à effectuer. On risque de se retrouver en revanche avec une table dont la structure est peu compréhensible.

Enfin la troisième solution (table reflétant exactement chaque classe de la hiérarchie, avec jointure(s) pour

reconstituer l'information) est la plus séduisante intellectuellement (de mon point de vue). Il n'y a pas de redondance, et il est facile d'ajouter de nouvelles sous-classes. L'inconvénient principal est la nécessité d'effectuer autant de jointures qu'il existe de niveaux dans la hiérarchie des classes pour reconstituer un objet.

Nous aurons alors les trois tables suivantes :

— `Video (id_video, titre, annee)`
— `Film (id_video, genre, pays, id_realisateur)`
— `Reportage(id_video, lieu)`

Nous avons nommé les identifiants `id_video` pour mettre en évidence une contrainte qui n'apparaît pas clairement dans ce schéma (mais qui est spécificiable en SQL) : *comme un même objet est représenté dans les lignes de plusieurs tables, son identifiant est une valeur de clé primaire commune à ces lignes*.

Un exemple étant plus parlant que de longs discours, voici comment nous représentons deux objets *vidéos*, dont l'un est un un film et l'autre un reportage.

Tableau 6.1 – La table `Vidéo`

id_video	titre	année
1	Gravity	2013
2	Messner, profession alpiniste	2014

Rien n'indique dans cette table est la catégorie particulière des objets représentés. C'est conforme à l'approche objet : selon le point de vue on peut très bien se contenter de voir les objets comme instances de la super-classe. De fait, *Gravity* et *Messner* sont toutes deux des vidéos.

Voici maintenant la table *Film*, contenant la partie de la description de *Gravity* spécifique à sa nature de film.

Tableau 6.2 – La table `Film`

id_video	genre	pays	id_realisateur
1	Science-fiction	USA	59

Notez que l'identifiant de *Gravity* (la valeur de `id_video`) est le *même* que pour la ligne contenant le titre et l'année dans *Vidéo*. C'est logique puisqu'il s'agit du même objet. Dans *Film*, `id_video` est *à la fois* la clé primaire, et une clé étrangère référençant une ligne de la table *Video*. On voit facilement quelle requête SQL permet de reconstituer l'ensemble des informations de l'objet.

```
select * from Video as v, FilmV as f
where v.id_video=f.id_video
and titre='Gravity'
```

Dans le même esprit, voici la table *Reportage*.

Tableau 6.3 – La table `Reportage`

id_video	lieu
2	Tyroll du sud

En résumé, avec cette approche, l'information relative à un même objet est donc éparpillée entre différentes tables. Comme souligné ci-dessus, cela mène à une particularité originale : la clé primaire d'une table pour une sous-classe est *aussi* clé étrangère référençant une ligne dans la table représentant la super-classe.

6.4.5 Quiz

6.5 S5 : Un peu de rétro-ingénierie

Supports complémentaires :
- Diapositives: rétroingénierie
- Vidéo sur la rétro ingénierie

Pour bien comprendre le rapport entre un schéma entité-association et le schéma relationnel équivalent, il est intéressant de faire le chemin à l'envers, en partant d'un schéma relationnel connu et en reconstruisant la modélisation entité-association. Cette courte section présente cette démarche de *rétro-ingénierie* sur un de nos exemples.

6.5.1 Quelle conception pour la base des immeubles ?

Reprenons le schéma de la base des immeubles et cherchons à comprendre comment il est conçu. Ce schéma (relationnel) est le suivant.
- Immeuble (**id**, nom, adresse)
- Appart (**id** , no , surface , niveau , *idImmeuble*)
- Personne (**id**, prénom , nom , profession , *idAppart*)
- Propriétaire (**idPersonne , idAppart**, quotePart)

Pour déterminer le schéma E/A, commençons par trouver les types d'entité. Une entité se caractérise par un identifiant qui lui est propre et la rend indépendante. Trois types d'entité apparaissent clairement sur notre schéma : les immeubles, les appartements et les personnes.

Il reste donc la table *Propriétaire* dont l'identifiant est une paire constituée de l'identifiant d'un immeuble et de l'identifiant d'une personne. Cette structure de clé est caractéristique d'une association plusieurs-plusieurs entre les types *Immeuble* et *Personne*.

Finalement, nous savons que les associations plusieurs-un sont représentés dans le schéma relationnel par les clés étrangères. Un appartement est donc lié à un immeuble, une personne à un appartement. Celles de la table *Propriétaire* ont déjà été prises en compte puisqu'elles font partie de la représentation des associations plusieurs-plusieurs. On obtient donc le schéma de la Fig. 6.15.

6.5.2 Avec entités faibles

On pourrait, à partir de là, se poser des questions sur les choix de conception. Une possibilité intéressante en l'occurrence est d'envisager de modéliser les appartements par entité faible. Un appartement est en effet une entité qui est très fortement liée à son immeuble, et on peut très bien envisager un appartement comme un *composant* d'un immeuble, en l'identifiant relativement à cet immeuble. Le schéma E/A devient alors celui de la Fig. 6.16.

L'impact sur le schéma relationnel est limité, mais quand même significatif. La clé de la table *Appart* devient une paire *(idImmeuble, no)*, et les clés étrangères changent également. On obtient le schéma suivant.
- Immeuble (**id**, nom, adresse)
- Appart (**idImmeuble, no**, surface , niveau)

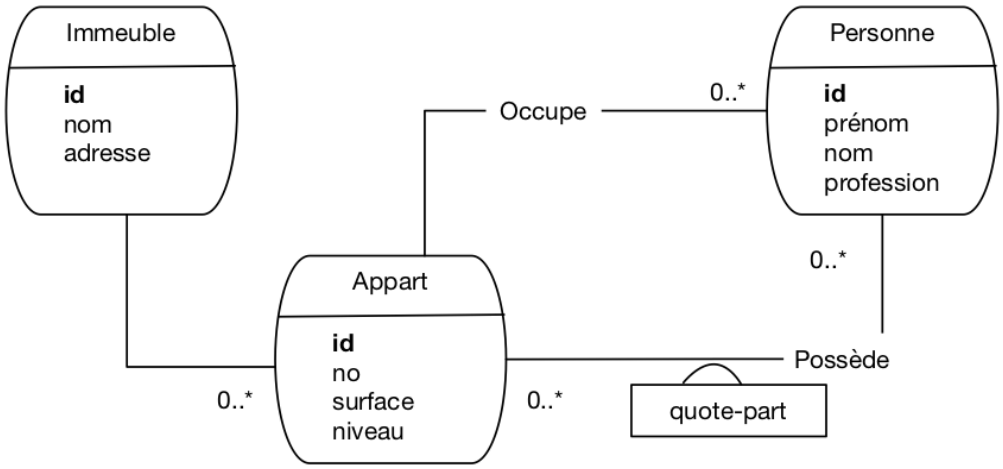

Fig. 6.15 – Le schéma E/A des immeubles après rétro-conception

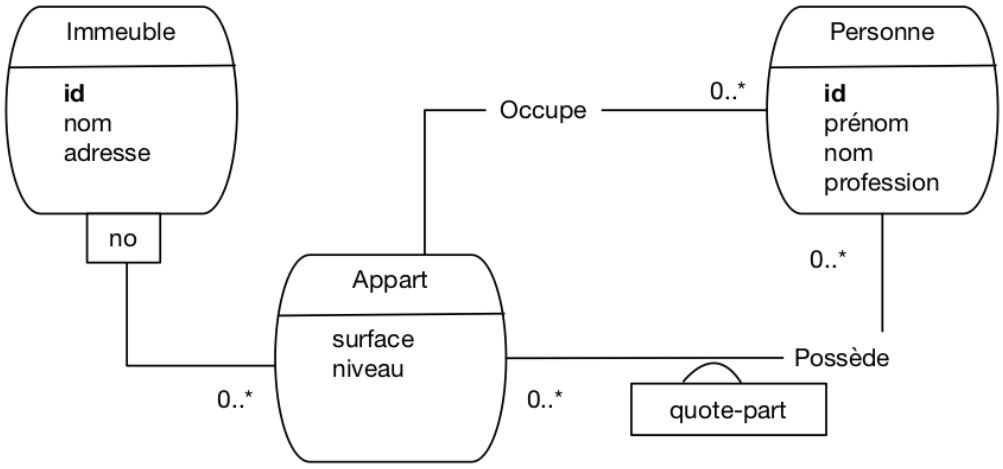

Fig. 6.16 – Le schéma E/A des immeubles avec entité faible

— Personne (**id**, prénom, nom , profession , *idImmeuble, no*)
— Propriété (**idPersonne, idImmeuble, no**, quotePart)

Il est important de noter que ce changement amènerait à modifier également les requêtes SQL des applications existantes. *Tout changement affectant une clé a un impact sur l'ensemble du système d'information*. D'où l'intérêt (et même l'impératif) de bien réfléchir avant de valider une conception.

6.5.3 Avec réification

On pourrait aussi réifier l'association plusieurs-plusieurs pour une faire un type d'entité *Propriétaire*. Le schéma entité-association est donné par la Fig. 6.17.

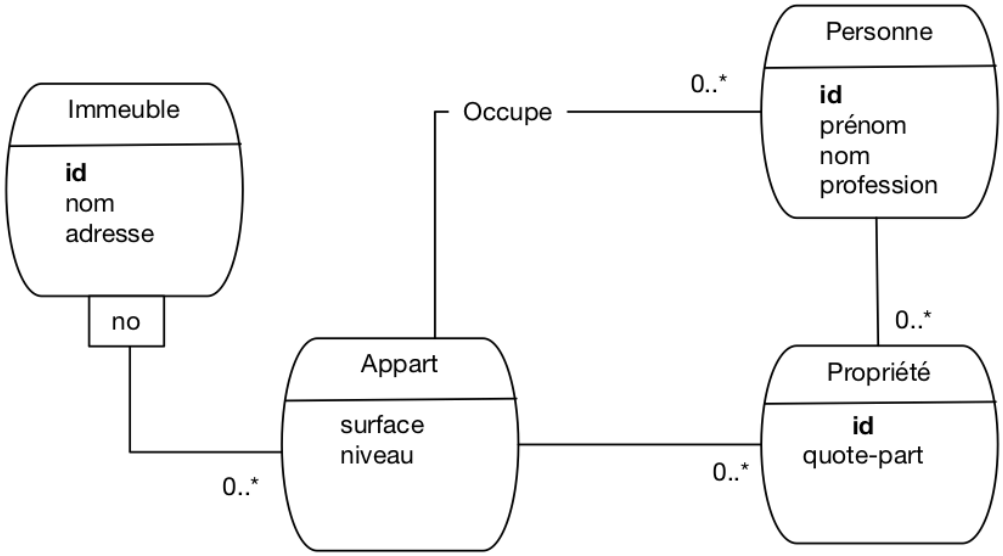

Fig. 6.17 – Le schéma E/A des immeubles avec réification

L'impact principal est le changement de la clé de la table *Propriétaire* qui devient un identifiant propre. Voici le schéma relationnel.

— Immeuble (**id**, nom, adresse)
— Appart (**idImmeuble, no** , surface , niveau)
— Personne (**id**, prénom , nom , profession , *idImmeuble, no*)
— Propriété (**id**, *idPersonne* , *idImmeuble, no*, quotePart)

Une conséquence importante est que la contrainte d'unicité sur le triplet *(idImmeuble, no, idPersonne)* disparaît. On pourrait donc avoir plusieurs nuplets pour la même personne avec le même immeuble. Dans ce cas précis, cela ne semble pas souhaitable et on peut alors déclarer que ce triplet *(idImmeuble, no, idPersonne)* est une clé candidate et lui associer une contrainte d'unicité (nous verrons ultérieurement comment). On peut aussi en conclure que la réification n'apporte rien en l'occurrence et conserver la modélisation initiale.

Voici un petit échantillon des choix à effectuer pour concevoir une base de données viable à long terme !

6.6 Exercices

Note : vous pouvez produire des diagrammes assez facilement avec https://www.lucidchart.com

---

Exercice Ex-conc-1 : compréhension d'un schéma E/A

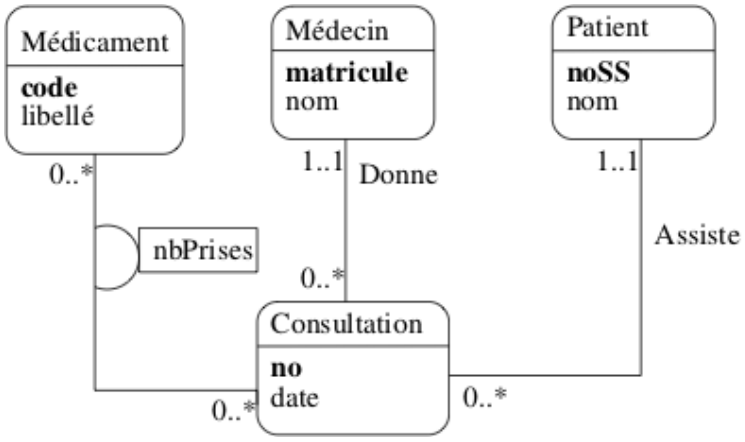

Fig. 6.18 – Un schéma pour un centre médical

On vous donne un schéma E/A (Fig. 6.18) représentant des visites dans un centre médical. Répondez aux questions suivantes *en fonction des caractéristiques de ce schéma* (autrement dit, indiquez si la situation décrite est représentable avec ce schéma, indépendamment de sa vraissemblance).

- Un patient peut-il effectuer plusieurs consultations ?
- Un médecin peut-il recevoir plusieurs patients dans la même consultation ?
- Peut-on prescrire plusieurs médicaments dans une même consultation ?
- Deux médecins différents peuvent-ils prescrire le même médicament ?
- Un patient peut-il voir deux médecins dans la même consultation ?
- Existe-t-il une posibilité de réifier une des associations, si oui laquelle ?
- Un médecin peut sans doute être également un patient, et vice-versa. Que devient le schéma avec cette hypothèse ?

Exercice Ex-conc-2 : un quotidien

Voici le schéma E/A (Fig. 6.19) du système d'information (très simplifié) d'un quotidien.

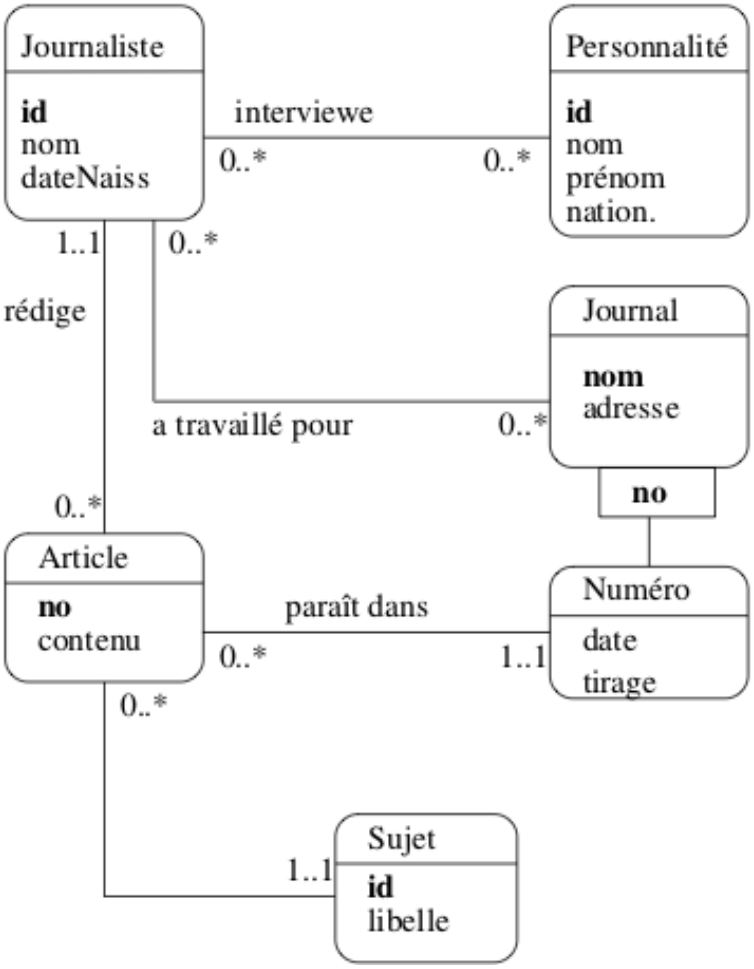

Fig. 6.19 – Système d'information d'un quotidien

Répondez aux questions suivantes, en fonction des caractéristiques du modèle.

- Un article peut-il être rédigé par plusieurs journalistes ?
- Un article peut-il être publié plusieurs fois ?
- Peut-il y avoir plusieurs articles sur le même sujet dans le même numéro ?
- Connaissant un article, est-ce que je connais le journal dans lequel il est paru ?
- Un journaliste peut-il travailler pour deux journaux en même temps ?
- Voyez-vous une entité faible ? Comment justifieriez-vous ce choix ?

Exercice Ex-conc-3 : cardinalités

Voici (Fig. 6.20) le début d'un schéma E/A pour la gestion d'une médiathèque. La spécification des besoins est la suivante : un disque est constitué d'un ensemble de plages. Chaque plage contient un oeuvre et une seule, mais une œuvre peut s'étendre sur plusieurs plages (par exemple une symphonie en 4 mouvements). De plus, pour chaque plage, on connaît les interprètes.

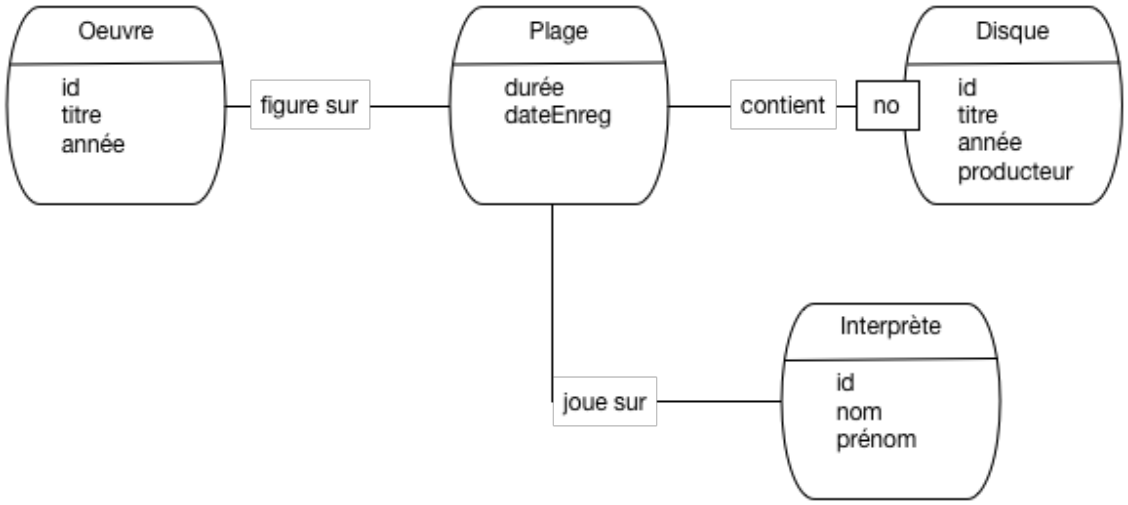

Fig. 6.20 – Début de schéma pour une médiathèque

Répondez aux questions suivantes

- Complétez le modèle de la Fig. 6.20 en ajoutant les cardinalités.
- On suppose que chaque interprète utilise un instrument (voix, piano, guitare, etc) et un seul sur une plage. Où placer l'attribut « Instrument » dans le modèle précédent ?
- Transformez l'association « Joue » dans la Fig. 6.20 en entité (réification). Donnez le nouveau modèle, sans oublier les cardinalités.
- Introduisez maintenant un entité Auteur (d'une œuvre) dans le schéma. Une œuvre peut avoir plusieurs auteurs.

Exercice Ex-conc-4 : normalisation

Reprenons le schéma de la Fig. 6.18. Nous allons appliquer l'algorithme de normalisation.

- Indiquez toutes les dépendances fonctionnelles définies par ce schéma
- En déduire le schéma des relations, en mettant en valeur clés primaires et clés étrangères

---

Exercice Ex-conc-5 : schéma relationnel

Donner le schéma relationnel pour le modèle de la médiathèque (partez du modèle initial, celui de la Fig. 6.20 ou, mieux, de votre modèle final.

Exercice Ex-conc-6 : rétroconception

Vous connaissez maintenant par cœur le schéma de la base des voyageurs. Saurez-vous trouver, par inversion de l'algorithme de normalisation, le schéma E/A dont il est issu ?

6.7 Atelier : étude du cas « Zoo » : le schéma normalisé

Nous reprenons l'étude du schéma « Zoo » vu dans le chapitre *Le modèle relationnel*. Maintenant, on va chercher un schéma correct.

— Donner une décomposition en troisième forme normale.
— Quelles sont, dans la décomposition, les clés primaires et étrangères.
— Quel serait le schéma E/A correspondant ?
— Est-il encore possible d'avoir les anomalies constatées dans la table initiale ?
— Pour conclure, vous pouvez (optionnel) installer un outil de conception comme le *MySql workbench* (https://www.mysql.com/fr/products/workbench/) et saisir votre schéma entité association avec le module *Entity relationship diagram* (voir Fig. 6.21). À partir de là vous pouvez tout paramétrer et engendrer automatiquement le schéma de la base.

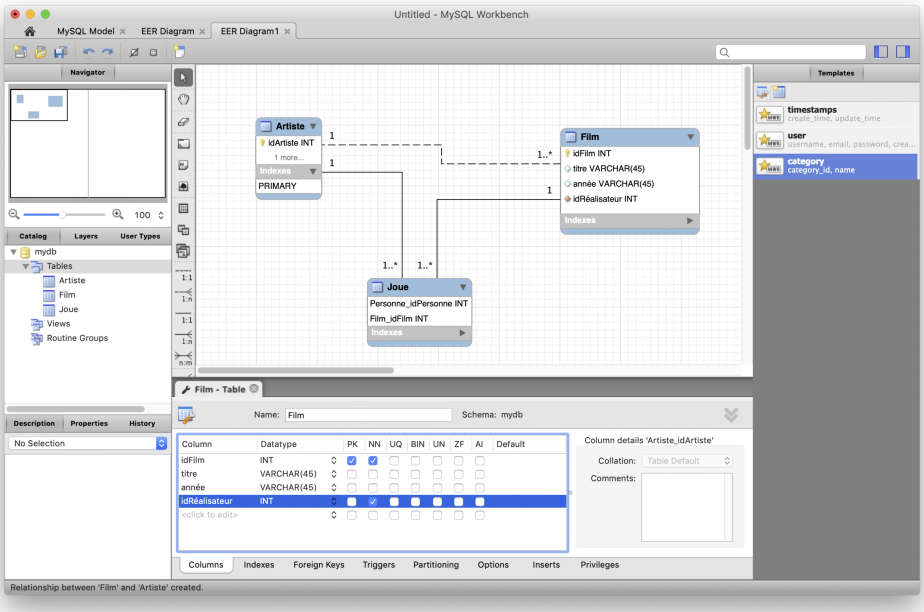

Fig. 6.21 – L'outil de conception de MySQL

6.8 Atelier collectif : concevons le système d'information d'un hôpital

À faire en groupe : proposer le modèle conceptuel d'une base de données permettant à un hôpital de gérer l'admission de ses patients, leur prise en charge par des médecins, leur placement dans une chambre, les examens qui leur sont prescrits, etc. Pas de spécification particulière : les besoins sont à exprimer en groupe, et la solution aux besoins s'élabore par consensus.

Vous pouvez utiliser un outil comme Mysqlworkbench, ou travailler sur papier ou au tableau. À la fin, il faut disposer d'un schéma relationnel correct : en troisième forme normale, et correspondant aux besoins.

CHAPITRE 7

Schémas relationnel

Ce chapitre présente le *langage de définition de données* (LDD) qui permet de spécifier le schéma d'une base de données relationnelle. Ce langage correspond à une partie de la norme SQL (*structured query language*), l'autre partie étant relative à la *manipulation des données* (LMD).

La définition d'un schéma comprend essentiellement deux parties : d'une part la description des *tables*, d'autre part les *contraintes* qui portent sur leur contenu. La spécification des contraintes est souvent placée au second plan bien qu'elle soit en fait très importante : elle permet d'assurer, *au niveau de la base*, des contrôles sur l'intégrité des données qui s'imposent à toutes les applications accédant à cette base. Un dernier aspect de la définition d'un schéma, rapidement survolé ici, est la description de la représentation dite « physique », celle qui décrit l'organisation des données. Il est toujours possible de réorganiser une base, et on peut donc tout à fait adopter initialement l'organisation choisie par défaut pour le système.

7.1 S1 : Création d'un schéma SQL

Supports complémentaires :

- Diapositives: spécification d'un schéma relationnel
- Vidéo sur la spécification d'un schéma relationnel / association

Passons aux choses concrètes : vous avez maintenant un serveur de base de données en place, vous disposez d'un compte d'accès, vous avez conçu votre base de données et vous voulez concrètement la mettre en œuvre. Nous allons prendre pour fil directeur la base des films. La première chose à faire est de créer une base spécifique avec la commande suivante :

```
create database Films
```

Il est d'usage de créer un utilisateur ayant les droits d'administration de cette base.

```
grant all on Films.* to philippe identified by 'motdepasse'
```

Voilà, maintenant il est possible d'ouvrir une connexion à la base `Film` sous le compte `philippe` et de créer notre schéma.

7.1.1 Types SQL

La norme SQL ANSI propose un ensemble de types dont les principaux sont donnés dans le tableau ci-dessous. Ce tableau présente également la taille, en octets, des instances de chaque type, cette taille n'étant ici qu'à titre indicatif car elle peut varier selon les systèmes.

Type	Description	Taille
integer	Type des entiers relatifs	4 octets
smallint	idem	2 octets
bigint	idem	8 octets
float	Flottants simple précision	4 octets
double	Flottants double précision	8 octets
real	Flottant simple ou double	8 octets
numeric (*M, D*)	Numérique avec précision fixe.	*M* octets
decimal(*M, D*)	Idem.	*M* octets
char(*M*)	Chaînes de longueur fixe	*M* octets
varchar*(M*)	Chaînes de longueur variable	*L+1* avec $L \leq M$
bit varying	Chaînes d'octets	Longueur de la chaîne.
date	Date (jour, mois, an)	env. 4 octets
time	Horaire (heure, minutes, secondes)	env. 4 octets
datetime	Date et heure	8 octets
year	Année	2 octets

Types numériques exacts

La norme SQL ANSI distingue deux catégories d'attributs numériques : les *numériques exacts*, et les *numériques flottants*. Les types de la première catégorie (essentiellement `integer` et `decimal`) permettent de spécifier la précision souhaitée pour un attribut numérique, et donc de représenter une valeur exacte. Les numériques flottants correspondent aux types couramment utilisés en programmation (`float`, `double`) et ne représentent une valeur qu'avec une précision limitée.

Le type `integer` permet de stocker des entiers, sur 4 octets. Il existe deux variantes du type `integer` : `smallint` et `bigint`. Ces types diffèrent par la taille utilisée pour le stockage : voir le tableau des types SQL.

Le type `decimal(M, D)` correspond à un numérique de taille maximale `M`, avec un nombre de décimales fixé à `D`. `numeric` est un synonyme de `decimal`. Ces types sont surtout utiles pour manipuler des valeurs dont la précision est connue, comme les valeurs monétaires. Afin de préserver cette précision, les instances de ces types sont stockées comme des chaînes de caractères.

Types numériques flottants

Ces types s'appuient sur la représentation des numériques flottants propre à la machine, en simple ou double précision. Leur utilisation est donc analogue à celle que l'on peut en faire dans un langage de programmation comme le C.

— Le type `float` correspond aux flottants en simple précision.
— Le type `double precision` correspond aux flottants en double précision ; le raccourci `double` est accepté.

Caractères et chaînes de caractères

Les deux types principaux de la norme ANSI sont `char` et `varchar`. Ces deux types permettent de stocker des chaînes de caractères d'une taille maximale fixée par le paramètre `M`. Les syntaxes sont identiques. Pour le premier, `char(M)`, et `varchar(M)` pour le second. La différence essentielle est qu'une valeur `char` a une taille fixée, et se trouve donc complétée avec des blancs si sa taille est inférieure à `M`. En revanche une valeur `varchar` a une taille variable et est tronquée après le dernier caractère non blanc.

Quand on veut stocker des chaînes de caractères longues (des textes, voire des livres), dont la taille dépasse, typiquement, 255 caractères, le type `varchar` ne suffit plus. La norme SQL propose un type `bit varying` qui correspond à de très longues chaînes de caractères. Souvent les systèmes proposent des variantes de ce type sous le nom `text` ou `blob` (pour *Binary Long Object*).

Dates

Un attribut de type `date` stocke les informations jour, mois et année (sur 4 chiffres). La représentation interne n'est pas spécifiée par la norme. Tous les systèmes proposent de nombreuses opérations de conversion (non normalisées) qui permettent d'obtenir un format d'affichage quelconque.

Un attribut de type `time` représente un horaire avec une précision à la seconde. Le type `datetime` permet de combiner une date et un horaire.

7.1.2 Création des tables

D'une manière générale, les objets du schéma sont créés avec `create`, modifiés avec `alter` et détruits avec `drop`, alors que les données, instances du schéma sont créées, modifiées et détruites avec, respectivement, `insert`, `update` et `delete`.

Voici un premier exemple avec la commande de création de la table *Internaute*.

```
create table Internaute (email varchar (40) not null,
                         nom varchar (30) not null ,
                         prénom varchar (30) not null,
                         région varchar (30),
                         primary key (email));
```

La syntaxe se comprend aisément. La seule difficulté est de choisir correctement le type de chaque attribut.

Conventions : noms des tables, des attributs, mots-clé SQL

On dispose, comme dans un langage de programmation, d'une certaine liberté. La seule recommandation est d'être cohérent pour des raisons de lisibilité. D'une manière générale, SQL n'est pas sensible à la casse. Quelques propositions :

— Le nom des tables devrait commencer par une majuscule, le nom des attributs par une minuscule ;

— quand un nom d'attribut est constitué de plusieurs mots, on peut soit les séparer par des caractères "_", soit employer la convention *CamelCase* : minuscule au premier mot, majuscule aux suivants. Exemple : `mot_de_passe` ou `motDePasse`.
— Majuscule ou minuscule pour les mots-clé SQL ? Quand on inclut une commande SQL dans un langage de programmation, il est peut-être plus lisible d'utiliser des majuscules pour les mots-clé.
— Les accents et caractères diacritiques sont-ils acceptés ? En principe oui, c'est ce que nous faisons ici. Cela implique de pouvoir aussi utiliser des accents dans les programmes qui incluent des commandes SQL et donc d'utiliser un encodage de type UTF8. Il faut vérifier si c'est possible dans l'environnement de développement que vous utilisez. Dans le doute, il vaut peut-être mieux sacrifier les accents.

Le `not null` dans la création de table *Internaute* indique que l'attribut correspondant doit *toujours* avoir une valeur. Il s'agit d'une différence importante entre la pratique et la théorie : on admet que certains attributs peuvent ne pas avoir de valeur, ce qui est très différent d'une chaîne vide ou de 0. Il est préférable d'ajouter la contrainte `not null` quand c'est pertinent : cela renforce la qualité de la base et facilite le travail des applications par la suite. L'option suivante permet ainsi de garantir que tout internaute a un mot de passe.

```
motDePasse  varchar(60) not null
```

Le SGBD rejettera alors toute tentative d'insérer un nuplet dans *Internaute* sans donner de mot de passe.

Important : La clé primaire doit *toujours* être déclarée `not null`.

Une autre manière de forcer un attribut à toujours prendre une valeur est de spécifier une *valeur par défaut* avec l'option `default`.

```
create table Cinéma (id integer not null,
                     nom varchar (30) not null ,
                     adresse varchar(255) default 'Inconnue',
                     primary key (id));
```

Quand on insérera un nuplet dans la table *Cinéma* sans indiquer d'adresse, le système affectera automatiquement la valeur `'Inconnue'` à cet attribut. En général on utilise comme valeur par défaut une constante, sauf pour quelques variables fournies par le système (par exemple `sysdate` pour indiquer la date courante).

7.1.3 Contraintes

La création d'une table telle qu'on l'a vue précédemment est assez sommaire car elle n'indique que le contenu de la table sans spécifier les contraintes que doit respecter ce contenu. Or il y a *toujours* des contraintes et il est indispensable de les inclure dans le schéma pour assurer (dans la mesure du possible) l'intégrité de la base.

Voici les règles (ou *contraintes d'intégrité*) que l'on peut demander au système de garantir :
— La valeur d'un attribut doit être unique au sein de la table.
— Un attribut doit toujours avoir une valeur. C'est la contrainte `not null` vue précédemment.
— Un attribut (ou un ensemble d'attributs) constitue(nt) la clé de la table.
— Un attribut dans une table est liée à la clé primaire d'une autre table (*intégrité référentielle*).

— Enfin toute règle s'appliquant à la valeur d'un attribut (min et max par exemple).

Les contraintes sur les clés (unicité et intégrité référentielle) doivent être systématiquement spécifiées.

Clés d'une table

Il peut y avoir plusieurs clés dans une table (les clés « candidates ») mais l'une d'entre elles doit être choisie comme *clé primaire*. Ce choix est important : la clé primaire est la clé utilisée pour référencer un nuplet et un seul à partir d'autres tables. Il est donc très délicat de la remettre en cause après coup. En revanche les clés secondaires peuvent être créées ou supprimées beaucoup plus facilement.

La clé primaire est spécifiée avec l'option `primary key`.

```
create table Pays (code varchar(4) not null,
                   nom  varchar (30) not null,
                   langue varchar (30) not null,
                   primary key (code));
```

Il doit *toujours* y avoir une clé primaire dans une table. Elle sert à garantir l'absence de doublon et à désigner un nuplet de manière univoque. Une clé peut être constituée de plusieurs attributs :

```
create table Notation (idFilm integer not null,
                       email  varchar (40) not null,
                       note  integer not null,
                       primary key (idFilm, email));
```

Tous les attributs figurant dans une clé doivent être déclarés `not null`. Cela n'a pas de sens d'identifier des nuplets par des valeurs absentes.

Comme nous l'avons déjà expliqué à plusieurs reprises, la méthode recommandée pour gérer la clé primaire est d'utiliser un attribut `id`, sans aucune signification particulière autre que celle de contenir la valeur unique identifiant un nuplet. Voici un exemple typique :

```
create table Artiste (id integer not null,
                      nom varchar (30) not null,
                      prénom varchar (30) not null,
                      annéeNaiss integer,
                      primary key (id))
```

La valeur de cet identifiant peut même est automatiquement engendrée à chaque insertion, ce qui soulage d'avoir à implanter un mécanisme de génération d'identifiant. La méthode varie d'un système à l'autre, et repose de manière générale sur la notion de *séquence*. Voici la syntaxe MySQL pour indiquer qu'une clé est auto-incrémentée.

```
create table Artiste (id integer not null auto increment,
                      ...,
                      primary key (id))
```

L'utilisation d'un identifiant artificiel n'apporte rien pour le contrôle des redondances. Il est possible d'insérer des centaines de nuplets dans la table *Artiste* ci-dessus ayant tous exactement les mêmes valeurs, et ne

différant que par la clé.

Les contraintes empêchant la redondance (et plus généralement assurant la cohérence d'une base) sont spécifiées indépendamment de la clé par la clause `unique`. On peut par exemple indiquer que deux artistes distincts ne peuvent avoir les mêmes nom et prénom.

```
create table Artiste  (idArtiste integer not null,
                       nom varchar (30) not null,
                       prénom varchar (30) not null,
                       annéeNaiss integer,
                       primary key (idArtiste),
                       unique (nom, prénom))
```

Il est facile de supprimer cette contrainte (dite de « clé secondaire ») par la suite. Ce serait beaucoup plus difficile si on avait utilisé la paire `(nom, prénom)` comme clé primaire puisqu'elle serait alors utilisée pour référencer un artiste dans d'autres tables.

La clause `unique` ne s'applique pas aux valeurs `null`.

7.1.4 Clés étrangères

SQL permet d'indiquer quelles sont les clés étrangères dans une table, autrement dit, quels sont les attributs qui font référence à un nuplet dans une autre table. On peut spécifier les clés étrangères avec l'option `foreign key`.

```
create table Film  (idFilm integer not null,
                    titre    varchar (50) not null,
                    année    integer not null,
                    idRéalisateur    integer not null,
                    genre varchar (20) not null,
                    résumé      varchar(255),
                    codePays    varchar (4),
                    primary key (idFilm),
                    foreign key (idRéalisateur) references Artiste(idArtiste),
                    foreign key (codePays) references Pays(code));
```

La commande

```
foreign key (idRéalisateur) references Artiste(idArtiste),
```

indique que `idRéalisateur` référence la clé primaire de la table *Artiste*. Le SGBD vérifiera alors, pour toute modification pouvant affecter le lien entre les deux tables, que la valeur de `idRéalisateur` correspond bien à un nuplet de *Artiste*. Ces modifications sont :

- l'insertion dans *Film* avec une valeur inconnue pour `idRéalisateur` ;
- la destruction d'un artiste ;
- la modification de `id` dans *Artiste* ou de `idRéalisateur` dans *Film*.

En d'autres termes on a la garantie que le lien entre *Film* et *Artiste* est *toujours* valide. Cette contrainte est importante pour s'assurer qu'il n'y a pas de fausse référence dans la base, par exemple qu'un film ne fait pas

référence à un artiste qui n'existe pas. Il est beaucoup plus confortable d'écrire une application par la suite quand on sait que les informations sont bien là où elles doivent être.

Il faut noter que l'attribut `codePays` n'est pas déclaré `not null`, ce qui signifie que l'on s'autorise à ne pas connaître le pays de production d'un film. Quand un attribut est à `null`, la contrainte d'intégrité référentielle ne s'applique pas. En revanche, on impose de connaître le réalisateur d'un film. C'est une contrainte forte, qui d'un côté améliore la richesse et la cohérence de la base, mais de l'autre empêche toute insertion, même provisoire, d'un film dont le metteur en scène est inconnu. Ces deux situations correspondent respectivement aux associations 0..* et 1..* dans la modélisation entité/association.

Note : On peut facilement passer un attribut de `not null` à `null`. L'inverse n'est pas vrai s'il existe déjà des valeurs à `null` dans la base.

Que se passe-t-il quand la violation d'une contrainte d'intégrité est détectée par le système ? Par défaut, la mise à jour est rejetée, mais il est possible de demander la répercussion de cette mise à jour de manière à ce que la contrainte soit respectée. Les événements que l'on peut répercuter sont la modification ou la destruction du nuplet référencé, et on les désigne par `on update` et `on delete` respectivement. La répercussion elle-même consiste soit à mettre la clé étrangère à `null` (option `set null`), soit à appliquer la même opération aux nuplets de l'entité composante (option `cascade`).

Voici comment on indique que la destruction d'un pays déclenche la mise à `null` de la clé étrangère `codePays` pour tous les films de ce pays.

```
create table Film  (idFilm integer not null,
                    titre    varchar (50) not null,
                    année    integer not null,
                    idRéalisateur    integer not null,
                    genre varchar (20) not null,
                    résumé      varchar(255),
                    codePays    varchar (4),
                    primary key (idFilm),
                    foreign key (idRéalisateur) references Artiste(idArtiste),
                    foreign key (codePays) references Pays(code)
                       on delete set null)
```

Dans le cas d'une entité faible, on décide en général de détruire le *composant* quand on détruit le *composé*. Par exemple, quand on détruit un cinéma, on veut également détruire les salles ; quand on modifie la clé d'un cinéma, on veut répercuter la modification sur ses salles (la modification d'une clé est très déconseillée, mais malgré tout autorisée). Dans ce cas c'est l'option `cascade` qui s'impose.

```
create table Salle  (idCinéma integer not null,
                     noSalle    integer not null,
                     capacité    integer not null,
                     primary key (idCinéma, noSalle),
                     foreign key (idCinéma) references Cinéma(idCinéma)
                       on delete cascade,
                       on update cascade)
```

L'attribut `idCinema` fait partie de la clé et ne peut donc pas être `null`. On ne pourrait donc pas spécifier ici `on delete set null`.

La spécification des actions `on delete` et `on update` simplifie la gestion de la base par la suite : on n'a plus par exemple à se soucier de détruire les salles quand on détruit un cinéma.

7.1.5 Quiz

7.2 S2 : Compléments

Supports complémentaires :

Pas de vidéo pour cette session qui présente quelques commandes utilitaires.

7.2.1 La clause `check`

La clause `check` exprime des contraintes portant soit sur un attribut, soit sur un nuplet. La condition elle-même peut être toute expression suivant la clause `where` dans une requête SQL. Les contraintes les plus courantes sont celles consistant à restreindre un attribut à un ensemble de valeurs, comme expliqué ci-dessous. On peut trouver des contraintes arbitrairement complexes, faisant référence à d'autres tables.

Voici un exemple simple qui restreint les valeurs possibles des attributs `année` et `genre` dans la table *Film*.

```
create table Film  (idFilm integer not null,
                    titre    varchar (50) not null,
                    année    integer
                       check (année between 1890 and 2020) not null,
                    idRéalisateur    integer,
                    genre varchar (20) l
                          check (genre in ('Histoire','Western','Drame')) not␣
↪null,
                    résumé      varchar(255),
                    codePays    varchar (4),
                    primary key (idFilm),
                    foreign key (idRéalisateur) references Artiste,
                    foreign key (codePays) references Pays)
```

Au moment d'une insertion dans la table *Film*, ou d'une modification de l'attribut `année` ou `genre`, le SGBD vérifie que la valeur insérée dans `genre` appartient à l'ensemble énuméré défini par la clause `check`.

Une autre manière de définir, dans la base, l'ensemble des valeurs autorisées pour un attribut – en d'autres termes, une codification imposée – consiste à placer ces valeurs dans une table et la lier à l'attribut par une contrainte de clé étrangère. C'est ce que nous pouvons faire par exemple pour la table *Pays*.

```
create table Pays (code    varchar(4) not null,
                   nom  varchar (30) default 'Inconnu' not null,
                  langue varchar (30) not null,
                  primary key (code));
 insert into Pays (code, nom, langue) values ('FR', 'France', 'Français');
 insert into Pays (code, nom, langue) values ('USA', 'Etats Unis', 'Anglais');
 insert into Pays (code, nom, langue) values ('IT', 'Italie', 'Italien');
 insert into Pays (code, nom, langue) values ('GB', 'Royaume-Uni', 'Anglais');
 insert into Pays (code, nom, langue) values ('DE', 'Allemagne', 'Allemand');
 insert into Pays (code, nom, langue) values ('JP', 'Japon', 'Japonais');
```

Si on ne fait pas de vérification automatique, soit avec `check`, soit avec la commande `foreign key`, il faut faire cette vérification dans l'application, ce qui est plus lourd à gérer.

7.2.2 Modification du schéma

La création d'un schéma n'est qu'une première étape dans la vie d'une base de données. On est toujours amené par la suite à créer de nouvelles tables, à ajouter des attributs ou à en modifier la définition. La forme générale de la commande permettant de modifier une table est :

```
alter table <nomTable> <action> <description>
```

où `action` peut être principalement `add`, `modify`, `drop` ou `rename` et `description` est la commande de modification associée à `action`. La modification d'une table peut poser des problèmes si elle est incompatible avec le contenu existant. Par exemple passer un attribut à `not null` implique que cet attribut a déjà des valeurs pour tous les nuplets de la table.

Modification des attributs

Voici quelques exemples d'ajout et de modification d'attributs. On peut ajouter un attribut `region` à la table *Internaute* avec la commande :

```
alter table Internaute add région varchar(10)
```

S'il existe déjà des données dans la table, la valeur sera à `null` ou à la valeur par défaut. La taille de `région` étant certainement insuffisante, on peut l'agrandir avec `modify`, et la déclarer `not null` par la même occasion :

```
alter table Internaute modify région varchar(30) not null
```

Il est également possible de diminuer la taille d'un attribut, avec le risque d'une perte d'information pour les données existantes. On peut même changer son type, pour passer par exemple de `varchar` à `integer`, avec un résultat imprévisible.

La commande `alter table` permet d'ajouter une valeur par défaut.

```
alter table Internaute add région set default 'Corse'
```

Enfin, on peut détruire un attribut avec drop.

```
alter table Internaute drop région
```

De plus, chaque système propose des commandes non normalisées. MySQL par exemple dispose d'une commande truncate pour « vider » une table rapidement, sans effectuer de contrôle (!) À vous d'éplucher la documentation pour ces aspects spécifiques.

7.2.3 Création d'index

Pour compléter le schéma d'une table, on peut définir des *index*. Un index offre un chemin d'accès aux nuplets d'une table qui est considérablement plus rapide que le balayage de cette table – du moins quand le nombre de nuplets est très élevé. Les SGBD créent systématiquement un index sur la clé primaire de chaque table. Il y a plusieurs raisons à cela ;

- l'index permet de vérifier rapidement, au moment d'une insertion, que la clé n'existe pas déjà ;
- l'index permet également de vérifier rapidement la contrainte d'intégrité référentielle : la valeur d'une clé étrangère doit toujours être la valeur d'une clé primaire.
- beaucoup de requêtes SQL, notamment celles qui impliquent plusieurs tables (*jointure*), se basent sur les clés des tables pour reconstruire les liens. L'index peut alors être utilisé pour améliorer les temps de réponse.

Un index est également créé pour chaque clause unique utilisée dans la création de la table. On peut de plus créer d'autres index, sur un ou plusieurs attributs, si l'application utilise des critères de recherche autres que les clés primaire ou secondaires.

La commande pour créer un index est la suivante :

```
create [unique] index <nomIndex>  on <nomTable> (<attribut1> [, ...])
```

L'option unique indique qu'on ne peut pas trouver deux fois la même clé dans l'index. La commande ci-dessous crée un index de nom *idxNom* sur les attributs nom et prénom de la table *Artiste*. Cet index a donc une fonction équivalente à la clause unique déjà utilisée dans la création de la table.

```
create unique index idxNom  on Artiste (nom, prénom)
```

On peut créer un index, cette fois non unique, sur l'attribut genre de la table *Film*.

```
create index idxGenre on Film (genre)
```

Cet index permettra d'exécuter très rapidement des requêtes SQL ayant comme critère de recherche le genre d'un film.

```
select * from Film where genre = 'Western'
```

Cela dit il ne faut pas créer des index à tort et à travers, car ils ont un impact négatif sur les commandes d'insertion et de destruction. À chaque fois, il faut en effet mettre à jour tous les index portant sur la table, ce qui représente un coût certain.

Pour en savoir plus sur les index, et en général sur la gestion de l'organisation des données, je vous renvoie à la seconde partie du cours disponible à http://sys.bdpedia.fr.

7.3 S3 : Les vues

Supports complémentaires :
- Diapositives: les vues
- Vidéo sur les vues

Une requête SQL produit toujours une table. Cela suggère la possibilité d'ajouter au schéma des tables *calculées*, qui ne sont rien d'autre que le résultat de requêtes stockées. De telles tables sont nommées des *vues* dans la terminologie relationnelle. On peut interroger des vues comme des tables stockées et, dans certaines limites, faire des mises à jour des tables stockées au travers de vues.

Une vue n'induit aucun stockage puisqu'elle n'existe pas physiquement. Elle permet d'obtenir une représentation différente des tables sur lesquelles elle est basée avec deux grands avantages :
- on peut faciliter l'interrogation de la base en fournissant sous forme de vues des requêtes prédéfinies ;
- on peut masquer certaines informations en créant des vues et en forçant par des droits d'accès l'utilisateur à passer par ces vues pour accéder à la base.

Les *vues* constituent donc un moyen complémentaire de contribuer à la sécurité (par restriction d'accès) et à la facilité d'utilisation (en offrant une « schéma virtuel » simplifié).

7.3.1 Création et interrogation d'une vue

Une vue est en tout point comparable à une table : en particulier on peut l'interroger par SQL. La grande différence est qu'une vue est le résultat d'une requête avec la caractéristique essentielle que ce résultat est réévalué à chaque fois que l'on accède à la vue. En d'autres termes une vue est *dynamique* : elle donne une représentation fidèle de la base au moment de l'évaluation de la requête.

Une vue est essentiellement une requête à laquelle on a donné un nom. La syntaxe de création d'une vue est très simple :

```
create view nomvue ([listeattributs])
as          requete
[with check option]
```

Voici une vue sur la table *Immeuble* montrant uniquement le Koudalou.

```
create view Koudalou as
     select nom, adresse, count(*) as nb_apparts
     from Immeuble as i, Appart as a
     where i.nom='Koudalou'
     and i.id=a.idImmeuble
     group by i.id, nom, adresse
```

La destruction d'une vue a évidemment beaucoup moins de conséquences que pour une table puisqu'on supprime uniquement la *définition* de la vue pas son *contenu*.

On interroge la vue comme n'importe quelle table.

```
select * from Koudalou
```

nom	adresse	nb_apparts
Koudalou	3 rue des Martyrs	5

La vue fait maintenant partie du schéma. On ne peut d'ailleurs évidemment pas créer une vue avec le même nom qu'une table (ou vue) existante. La définition d'une vue peut consister en un requête SQL aussi complexe que nécessaire, avec jointures, regroupements, tris.

Allons un peu plus loin en définissant sous forme de vues un accès aux informations de notre base *Immeuble*, mais restreint uniquement à tout ce qui concerne l'immeuble Koudalou. On va en profiter pour offrir dans ces vues un accès plus facile à l'information. La vue sur les appartements, par exemple, va contenir contrairement à la table *Appart* le nom et l'adresse de l'immeuble et le nom de son occupant.

```
create or replace view AppartKoudalou as
   select no, surface, niveau, i.nom as immeuble, adresse,
          concat(p.prénom,  ' ', p.nom) as occupant
   from Immeuble as i, Appart as a, Personne as p
   where i.id=a.idImmeuble
   and   a.id=p.idAppart
   and   i.id=1
```

On voit bien sur cet exemple que l'un des intérêts des vues est de donner une représentation « dénormalisée » de la base en regroupant des informations par des jointures. Le contenu étant virtuel, il n'y a ici aucun inconvénient à « voir » la redondance du nom de l'immeuble et de son adresse. Le bénéfice, en revanche, est la possibilité d'obtenir très simplement toutes les informations utiles.

```
select * from AppartKoudalou
```

no	surface	niveau	immeuble	adresse	occupant
1	150	14	Koudalou	3 rue des Martyrs	Léonie Atchoum
51	200	2	Koudalou	3 rue des Martyrs	Barnabé Simplet
52	50	5	Koudalou	3 rue des Martyrs	Alice Grincheux
43	75	3	Koudalou	3 rue des Martyrs	Brandon Timide

Le nom des attributs de la vue est celui des expressions de la requête associée. On peut également donner ces noms après le `create view` à condition qu'il y ait correspondance univoque entre un nom et une expression du `select`. On peut ensuite donner des droits en lecture sur cette vue pour que cette information limitée soit disponible à tous.

```
grant select on Immeuble.Koudalou, Immeuble.AppartKoudalou to adminKoudalou
```

Pour peu que cet utilisateur n'ait aucun droit de lecture sur les tables de la base *Immeuble*, on obtient un moyen simple de masquer et restructurer l'information.

7.3.2 Mise à jour d'une vue

L'idée de modifier une vue peut sembler étrange puisqu'une vue n'a pas de contenu. En fait il s'agit bien entendu de modifier la table qui sert de support à la vue. Il existe de sévères restrictions sur les droits d'insérer ou de mettre à jour des tables au travers des vues. Un exemple suffit pour comprendre le problème. Imaginons que l'on souhaite insérer un nuplet dans la vue `AppartKoudalou`.

```
insert into AppartKoudalou (no, surface, niveau, immeuble, adresse, occupant)
values (1, 12, 4, 'Globe', '2 Avenue Leclerc', 'Palamède')
```

Le système rejettera cette requête (par exemple, pour MySQL, avec le message `Can not modify more than one base table through a join view 'Immeuble.AppartKoudalou'`). Cet ordre s'adresse à une vue issue de trois tables. Il n'y a clairement pas assez d'information pour alimenter ces tables de manière cohérente et l'insertion n'est pas possible (de même que toute mise à jour). De telles vues sont dites *non modifiables*. Les règles définissant les vues modifiables sont assez strictes et difficiles à résumer simplement d'autant qu'elles varient selon l'opération (`update`, `delete`, ou `insert`). En première approximation on peut retenir les points suivants qui donnent lieu à quelques exceptions sur lesquelles nous reviendrons ensuite.

— la vue doit être basée sur une seule table ;
— toute colonne non référencée dans la vue doit pouvoir être mise à `null` ou disposer d'une valeur par défaut ;
— on ne peut pas mettre à jour un attribut qui résulte d'un calcul ou d'une opération.

On ne peut donc pas insérer ou modifier la vue *Koudalou* à cause de la jointure et de l'attribut calculé. La requête suivante serait rejetée.

```
insert into Koudalou (nom, adresse)
values ('Globe', '2 Avenue Leclerc')
```

En revanche une vue portant sur une seule table avec un `select *` est modifiable.

```
create view PropriétaireAlice
   as select * from Propriétaire
  where idPersonne=2

insert into PropriétaireAlice values (2, 100, 20)
insert into PropriétaireAlice values (3, 100, 20)
```

Maintenant, si on fait :

```
select * from PropriétaireAlice
```

On obtient :

idPersonne	idAppart	quotePart
2	100	20
2	103	100

L'insertion précédente illustre une petite subtilité : on peut insérer dans une vue sans être en mesure de voir le nuplet inséré au travers de la vue par la suite ! On a en effet inséré dans la vue le propriétaire 3 qui est ensuite filtré quand on interroge la vue.

SQL propose l'option `with check option` qui permet de garantir que tout nuplet inséré dans la vue satisfait les critères de sélection de la vue.

```
create view PropriétaireAlice
as select * from Propriétaire
where idPersonne=2
with check option
```

SQL permet également la modification de vues définies par des jointures. Les restrictions sont essentielement les même que pour les vues mono-tabulaires : on ne peut insérer que dans une des tables (il faut donc préciser la liste des attributs) et tous les attributs `not null` doivent avoir une valeur. Voici un exemple de vue modifiable basée sur une jointure.

```
create or replace view ToutKoudalou
as select i.id as id_imm, nom, adresse, a.*
   from Immeuble as i, Appart as a
   where i.nom='oudalou'
   and i.id=a.idImmeuble
   with check option
```

Il est alors possible d'insérer à condition d'indiquer des attributs d'une seule des deux tables. La commande ci-dessous ajoute un nouvel appartement au *Koudalou*.

```
insert into ToutKoudalou (id, surface, niveau, idImmeuble, no)
values (104, 70, 12, 1, 65)
```

En conclusion, l'intérêt principal des vues est de permettre une restructuration du schéma en vue d'interroger et/ou de protéger des données. L'utilisation de vues pour des mises à jour devrait rester marginale.

7.3.3 Quiz

7.4 Exercices

Exercice Ex-schema-1 : le schéma du centre médical

Reprenons le centre médical étudié dans le chapitre *Conception d'une base de données*.

— Donnez les commandes SQL de création du schéma
— Donnez les commandes d'insertion pour une base avec un médecin (le Dr Folamour), un patient (M. Maboul), un médicament (le libellé est Trucmyl) et une consultation ayant donné lieu a la prescription de 3 prises de Trucmyl par le Dr Folamour à M. Maboul. Choisissez les identifiants à votre convenance.
— Créez une vue `DrFolamour` donnant pour toutes les prescriptions du Dr Folamour le nom du patient, le libellé du médicament et le nombre de prises

Exercice Ex-schema-2 : pour pratiquer

Donnez les commandes SQL de création du schéma pour le quotidien et la médiathèque.

7.5 Atelier : étude du cas « Zoo », suite et fin

Allons-y pour la création de la base (normalisée) de notre zoo.

- Donner les commandes de création des tables, avec clés primaire et clés étrangères.
- Ajoutez la contrainte suivante : deux animaux de la même espéce ne doivent pa avoir le même nom
- Reprendre le contenu de la table non-normalisée (ci-dessous), et donner les commandes d'insertion de ces données dans la base normalisée
- Exprimez les requêtes suivantes
  - Quels sont les ours du zoo ?
  - Quels animaux s'appellent Jojo ?
  - Quels animaux viennent de la planète Kashyyyk (quand ils ne sont pas prisonniers dans le zoo…) ?
  - De quels animaux s'occupe le gardien Jules ?
  - Sur quel(s) emplacement(s) y-a-il des animaux de classes différentes (donner aussi le nom du gardien)
  - Somme des salaires des gardiens.
  - Quels gardiens surveillent plus d'un emplacement
- Et pour finir, donnez la définition de la vue qui recrée, à partir de la base normalisée, le contenu de la table avant décomposition (ci-dessous).

codeAnimal	nom	espèce	gardien	salaire	classe	origine	codeEmplacement	surface
10	Zoé	Girafe	Marcel	20 000	Mammifère	Afrique	A	120
20	Chewbacca	Wookiee	Marcel	20 000		Kashyyyk	C	200
30	Jojo	Mérou	Jules	18 000	Poisson	Méditerranée	B	50
20	Irma	Perroquet	Marcel	20 000	Oiseau	Asie	A	120
40	Goupil	Renard	Jules	18 000	Insecte	Europe	B	50

À vous de jouer !

CHAPITRE 8

Procédures et déclencheurs

Le langage SQL n'est pas un langage de programmation au sens courant du terme. Il ne permet pas, par exemple, de définir des fonctions ou des variables, d'effectuer des itérations ou des instructions conditionnelles. Il ne s'agit pas d'un défaut dans la conception du langage, mais d'une orientation délibérée de SQL vers les opérations de recherche de données dans une base volumineuse, la priorité étant donnée à la *simplicité* et à *l'efficacité*. Ces deux termes ont une connotation forte dans le contexte d'un langage d'interrogation, et correspondent à des critères (et à des contraintes) précisément définis. La simplicité d'un langage est essentiellement relative à son caractère *déclaratif*, autrement dit à la capacité d'exprimer des recherches en laissant au système le soin de déterminer le meilleur moyen de les exécuter. L'efficacité est, elle, définie par des caractéristiques liées à la complexité d'évaluation sur lesquelles nous ne nous étendrons pas ici. Signalons cependant que la terminaison d'une requête SQL est *toujours* garantie, ce qui n'est pas le cas d'un programme écrit dans un langage plus puissant, .

Il est donc clair que SQL ne suffit pas pour le développement d'applications, et tous les SGBD relationnels ont, dès l'origine, proposé des interfaces permettant de l'associer à des langages plus classiques comme le C ou Java. Ces interfaces de programmation permettent d'utiliser SQL comme outil pour récupérer des données dans des programmes réalisant des tâches très diverses : interfaces graphiques, traitements « batch », production de rapports ou de sites web, etc. D'une certaine manière, on peut alors considérer SQL comme une interface d'accès à la base de données, intégrée dans un langage de programmation généraliste. Il s'agit d'ailleurs certainement de son utilisation la plus courante.

Pour certaines fonctionnalités, le recours à un langage de programmation « externe » s'avère cependant inadapté ou insatisfaisant. Une évolution des SGBD consiste donc à proposer, au sein même du système, des primitives de programmation qui viennent pallier le manque relatif d'expressivité des langages relationnnels. Le présent chapitre décrit ces évolutions et leur application à la création de *procédures stockées* et de déclencheurs (*triggers*). Les premières permettent d'enrichir un schéma de base de données par des calculs ou des fonctions qui ne peuvent pas - parfois même dans des cas très simples - être obtenus avec SQL ; les seconds étendent la possibilité de définir des contraintes.

Parallèlement à ces applications pratiques, les procédures stockées illustrent simplement les techniques d'in-

tégration de SQL à un langage de programmation classique, et soulignent les limites d'utilisation d'un langage d'interrogation, et plus particulièrement du modèle relationnel.

8.1 S1. Procédures stockées

Supports complémentaires :
- Diapositives: PL/SQL
- Vidéo sur PL/SQL

Comme mentionné ci-dessus, les procédures stockées constituent une alternative à l'écriture de programmes avec une langage de programmation généraliste. Commençons par étudier plus en détail les avantages et inconvénients respectifs des deux solutions avant d'entrer dans les détails techniques.

8.1.1 Rôle et fonctionnement des procédures stockées

Une procédure stockée s'exécute au sein du SGBD, ce qui évite les échanges réseaux qui sont nécessaires quand les mêmes fonctionnalités sont implantées dans un programme externe communiquant en mode client/serveur avec la base de données. La Fig. 8.1 illustre la différence entre les deux mécanismes. À gauche un programme externe, écrit par exemple en Java ou en Python, doit tout d'abord se connecter au serveur du SGBD. Le programme s'exécute alors en communiquant avec le serveur pour exécuter les requêtes et récupérer les résultats. Dans cette architecture, chaque demande d'exécution d'un ordre SQL implique une transmission sur le réseau, du programme vers le client, suivie d'une analyse de la requête par le serveur, de sa compilation et de son exécution (Dans certains cas les requêtes du programme client peuvent être pré-compilées, ou « préparées »). Ensuite, chaque fois que le programme client souhaite récupérer un n-uplet du résultat, il doit effectuer un appel externe, via le réseau. Tous ces échanges interviennent de manière non négligeable dans la performance de l'ensemble, et cet impact est d'autant plus élevé que les communications réseaux sont lentes et/ou que le nombre d'appels nécessaires à l'exécution du programme est important.

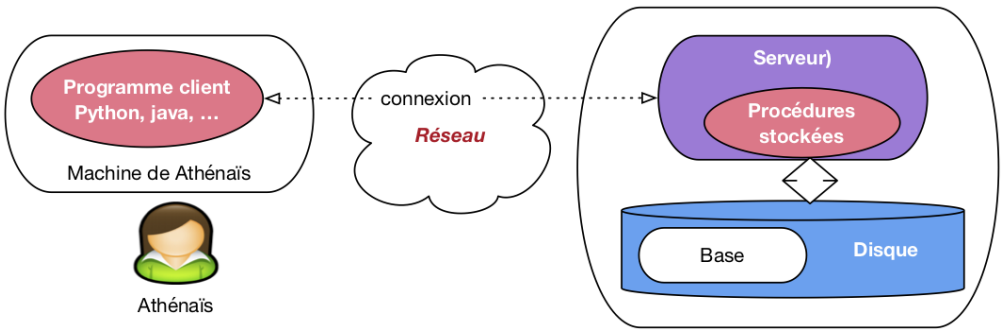

Fig. 8.1 – Comparaison programmes externes/procédures stockées

Le recours à une procédure stockée permet de regrouper du côté serveur l'ensemble des requêtes SQL et le traitement des données récupérées. La procédure est compilée une fois par le SGBD, au moment de sa création, ce qui permet de l'exécuter rapidement au moment de l'appel. De plus les échanges réseaux ne sont plus nécessaires puisque la logique de l'application est étroitement intégrée aux requêtes SQL. Le rôle du

programme externe se limite alors à se connecter au serveur et à demander l'exécution de la procédure, en lui passant au besoin les paramètres nécessaires.

Bien entendu, en pratique, les situations ne sont pas aussi tranchées et le programme externe est en général amené à appeler plusieurs procédures, jouant en quelque sorte le rôle de coordinateur. Si les performances du système sont en cause, un recours judicieux aux procédures stockées reste cependant un bon moyen de réduire le trafic client-serveur.

L'utilisation de procédures stockées est par ailleurs justifiée, même en l'absence de problèmes de performance, pour des fonctions très « sensibles », terme qui recouvre (non exclusivement) les cas suivants :

1. la fonction est basée sur des règles complexes qui doivent être implantées très soigneusement ;
2. la fonction met à jour des données dont la correction et la cohérence sont indispensable au bon fonctionnement de l'application ;
3. la fonction évolue souvent.

On est souvent amené, quand on développe une application, à utiliser plusieurs langages en fonction du contexte : le langage C ou Java pour les traitements *batch*, PHP ou Python pour l'interface web, un générateur d'application propriétaire pour la saisie et la consultation à l'écran, un langage de script pour la production de rapports, etc. Il est important alors de pouvoir factoriser les opérations de base de données partagées par ces différents contextes, de manière à les rendre disponibles pour les différents langages utilisés. Par exemple la réservation d'un billet d'avion, ou l'exécution d'un virement bancaire, sont des opérations dont le fonctionnement correct (pas deux billets pour le même siège ; pas de débit sans faire le crédit correspondant) et cohérent (les mêmes règles doivent être appliquées, quel que soit le contexte d'utilisation) doit toujours être assuré.

C'est facile avec une procédure stockée, et cela permet d'une part d'implanter une seule fois des fonctions « sensibles », d'autre part de garantir la correction, la cohérence et l'évolutivité en imposant l'utilisation systématique de ces fonctions au lieu d'un accès direct aux données.

Enfin, le dernier avantage des procédures stockées est la relative facilité de programmation des opérations de bases de données, en grande partie à cause de la très bonne intégration avec SQL. Cet aspect est favorable à la qualité et à la rapidité du développement, et aide également à la diffusion et à l'installation du logiciel puisque les procédures sont compilées par les SGBD et fonctionnent donc de manière identique sur toute les plateformes.

Il existe malheureusement une contrepartie à tous ces avantages : chaque éditeur de SGBD propose sa propre extension procédurale pour créer des procédures stockées, ce qui rend ces procédures incompatibles d'un système à un autre. Cela peut être dissuasif si on souhaite produire un logiciel qui fonctionne avec tous les SGBD relationnels.

La description qui suit se base sur le langage PL/SQL d'Oracle (« PL » signifie *Procedural Language*) qui est sans doute le plus riche du genre. Le même langage, simplifié, avec quelques variantes syntaxiques mineures, est proposé par PostgreSQL, et les exemples que nous donnons peuvent donc y être transposés sans trop de problème. Les syntaxes des langages utilisés par d'autres systèmes sont un peu différentes, mais tous partagent cependant un ensemble de concepts et une proximité avec SQL qui font de PL/SQL un exemple tout à fait représentatif de l'intérêt et de l'utilisation des procédures stockées.

8.1.2 Introduction à PL/SQL

Nous allons commencer par quelques exemples très simples, appliqués à la base *Films*, afin d'obtenir un premier aperçu du langage. Le premier exemple consiste en quelques lignes permettant d'afficher des statistiques sur la base de données (nombre de films et nombre d'artistes). Il ne s'agit pas pour l'instant d'une procédure stockée, mais d'un code qui est compilé et exécuté en direct.

```
-- Exemple de bloc PL/SQL donnant des informations sur la base

DECLARE
  -- Quelques variables
  v_nbFilms   integer;
  v_nbArtistes integer;

begin
  -- Compte le nombre de films
  select count(*) into v_nbFilms from Film;
  -- Compte le nombre d'artistes
  select count(*) into v_nbArtistes from Artiste;

  -- Affichage des résultats
  DBMS_outPUT.PUT_LinE ('Nombre de films: ' || v_nbFilms);
  DBMS_outPUT.PUT_LinE ('Nombre d''artistes: ' || v_nbArtistes);

  exception
    when others then
      DBMS_outPUT.PUT_LinE ('Problème rencontré dans StatsFilms');

end;
/
```

Le code est structuré en trois parties qui forment un « bloc » : déclarations des variables, instructions (entre `begin` et `end`) et gestion des exceptions. La première remarque importante est que les variables sont typées, et que les types sont exactement ceux de SQL (ou plus largement les types supportés par le SGBD, qui peuvent différer légèrement de la norme). Un autre aspect de l'intégration forte avec SQL est la possibilité d'effectuer des requêtes, d'utiliser dans cette requête des critères basés sur la valeur des variables de la procédure, et de placer le résultat dans une (ou plusieurs) variables grâce à la clause `into`. En d'autres termes on transfère directement des données représentées selon le modèle relationnel et accessibles avec SQL, dans des unités d'information manipulables avec les structures classiques (test ou boucles) d'un langage impératif.

Les fonctions SQL fournies par le SGBD sont également utilisables, ainsi que des librairies spécifiques à la programmation procédurale (des *packages* chez orACLE). Dans l'exemple ci-dessus on utilise le *package* `DBMS_outPUT` qui permet de produire des messages sur la sortie standard (l'écran en général).

La dernière section du programme est celle qui gère les « exceptions ». Ce terme désigne une erreur qui est soit définie par l'utilisateur en fonction de l'application (par exemple l'absence d'un n-uplet, ou une valeur incorrecte dans un attribut), soit engendrée par le système à l'exécution (par exemple une division par zéro, ou l'absence d'une table). Au moment où une erreur est rencontrée, PL/SQL redirige le flux d'exécution vers la section `exception` où le programmeur doit définir les actions à entreprendre au cas par cas. Dans

l'exemple précédent, on prend toutes les exceptions indifféremment (mot-clé `others` qui désigne le choix par défaut) et on affiche un message.

Note : Un programme Pl/SQL peut être placé dans un fichier et executé avec la commande `start` sous l'utilitaire de commandes. par exemple :

```
SQL>start StatsFilms
```

En cas d'erreur de compilation, la commande `SHOW ERRorS` donne la liste des problèmes rencontrés. Sinon le code est exécuté. Voici par exemple ce que l'on obtient avec le code donné précédemment en exemple (La commande `set serveroutput on` assure que les messages sont bien affichés à l'écran..

```
SQL> set serveroutput on
SQL> start StatsFilms
Nombre de films: 48
Nombre d'artistes: 126
```

Voici maintenant un exemple de procédure stockée. On retrouve la même structuration que précédemment (déclarations, instructions, exception), mais cette fois ce « bloc » est nommé, stocké dans la base au moment de la compilation, et peut ensuite être appelé par son nom. La procédure implante la règle suivante : l'insertion d'un texte dans la table des genres s'effectue toujours en majuscules, et on vérifie au préalable que ce code n'existe pas déjà.

```
-- Insère un nouveau genre, en majuscules, et en vérifiant
-- qu'il n'existe pas déjà

create or replace procedure InsereGenre (p_genre varchar) as

  -- Déclaration des variables
  v_genre_majuscules varchar(20);
  v_count integer;
  genre_existe exception;
begin
  -- On met le paramètre en majuscules
  v_genre_majuscules := UPPER(p_genre);

  -- On vérifie que le genre n'existe pas déjà
  select count(*) into v_count
  from Genre where code = v_genre_majuscules;

  -- Si on n'a rien trouvé: on insère
  if (v_count = 0) then
   insert into Genre (code) VALUES (v_genre_majuscules);
  else
   RAisE genre_existe;
 end if;
```

(suite sur la page suivante)

(suite de la page précédente)

```
  exception
    when genre_existe then
     DBMS_outPUT.PUT_LinE('Le genre existe déjà en ' ||
                          v_count || ' exemplaire(s).');

end;
/
```

La procédure accepte des paramètres qui, comme les variables, sont typées. Le corps de la procédure montre un exemple d'utilisation d'une fonction SQL fournie par le système, ici la fonction `UPPER` qui prend une chaîne de caractères en entrée et la renvoie mise en majuscules.

La requête SQL garantit que l'on obtient un et un seul n-uplet. Nous verrons plus loin comment traiter le cas où le résultat de la requête est une table contenant un nombre quelconque de n-uplets. Dans l'exemple ci-dessus on obtient toujours un attribut donnant le nombre de n-uplets existants dans la table pour le code que l'on veut insérer. Si ce nombre n'est pas nul, c'est que le genre existe déjà dans la table, et on produit une « exception » avec la clause `RAisE exception`, sinon c'est que le genre n'existe pas et on peut effectuer la clause d'insertion, en indiquant comme valeurs à insérer celles contenues dans les variables appropriées.

On peut appeler cette procédure à partir de n'importe quelle application connectée au SGBD. Sous SQL*Plus on utilise l'instruction `execute`. Voici par exemple ce que l'on obtient avec deux appels successifs.

```
SQL>  execute InsereGenre('Policier');

SQL>  execute InsereGenre('Policier');
Le genre existe déjà en 1 exemplaire(s).
```

Le premier appel s'est correctement déroulé puisque le genre « Policier » n'existait pas encore dans la table. Le second en revanche a échoué, ce qui a déclenché l'exception et l'affichage du message d'erreur.

On peut appeler `InsereGenre()` depuis un programme C, Java, ou tout autre outil. Si on se fixe comme règle de toujours passer par cette procédure pour insérer dans la table `Genre`, on est donc sûr que les contraintes implantées dans la procédure seront toujours vérifiées.

Note : Pour forcer les développeurs à toujours passer par la procédure, on peut fixer les droits d'accès de telle sorte que les utilisateurs orACLE aient le droit d'exécuter `InsereGenre()`, mais pas de droit de mise à jour sur la table `Genre` elle-même.

Voici un troisième exemple qui complète ce premier tour d'horizon rapide du langage. Il s'agit cette fois d'une *fonction*, la différence avec une procédure étant qu'elle renvoie une valeur, instance de l'un des types SQL. Dans l'exemple qui suit, la fonction prend en entrée l'identifiant d'un film et renvoie une chaîne de caractères contenant la liste des prénom et nom des acteurs du film, séparés par des virgules.

```
-- Fonction retournant la liste des acteurs pour un film donné

create or replace FUNCTIon MesActeurs(v_idFilm integer) RETURN varchar is
```

(suite sur la page suivante)

(suite de la page précédente)

```
  -- Le résultat
  resultat varchar(255);

begin
  -- Boucle prenant tous les acteurs du films

  for art in
    (select Artiste.* from Role, Artiste
    where idFilm = v_idFilm  and idActeur=idArtiste)
  loop

    if (resultat is NOT NULL) then
      resultat := resultat || ', ' || art.prenom || ' ' || art.nom;
    else
      resultat := art.prenom || ' ' || art.nom;
   end if;

  end loop;

 return resultat;
end;
/
```

La fonction effectue une requête SQL pour rechercher tous les acteurs du film dont l'identifiant est passé en paramètre. Contrairement à l'exemple précédent, cette requête renvoie en général plusieurs n-uplets. Une des caractéristiques principales des techniques d'accès à une base de données avec un langage procédural est que l'on ne récupère pas d'un seul coup le résultat d'un ordre SQL. Il existe au moins deux raisons à cela :

1. le résultat de la requête peut être extrêmement volumineux, ce qui poserait des problèmes d'occupation mémoire si on devait tout charger dans l'espace du programme client ;
2. les langages de programmation ne sont en général pas équipés nativement des types nécessaires à la représentation d'un ensemble de n-uplets.

Le concept utilisé, plus ou moins implicitement, dans toutes les interfaces permettant aux langages procéduraux d'accéder aux bases de données est celui de *curseur*. Un curseur permet de parcourir, à l'aide d'une boucle, l'ensemble des n-uplets du résultat d'une requête, en traitant le n-uplet courant à chaque passage dans la boucle. Ici nous avons affaire à la version la plus simple qui soit d'un curseur, mais nous reviendrons plus loin sur ce mécanisme.

Une fonction renvoie une valeur, ce qui permet de l'utiliser dans une requête SQL comme n'importe quelle fonction native du système. Voici par exemple une requête qui sélectionne le titre et la liste des acteurs du film dont l'identifiant est 5.

```
SQL> select titre, MesActeurs(idFilm)  from Film where idFilm=5;

TITRE        MESACTEURS(IDFILM)
------------ ------------------------------
Volte/Face   John Travolta, Nicolas Cage
```

On peut noter que le résultat de la fonction `MesActeurs()` ne peut pas être obtenu avec une requête SQL. Il est d'ailleurs intéressant de se demander pourquoi, et d'en tirer quelques conclusions sur certaines limites de SQL. Il est important de mentionner également qu'orACLE ne permet pas l'appel, dans un ordre `select` de fonctions effectuant des mises à jour dans la base : une requête n'est pas censée entraîner des modifications, surtout si elles s'effectuent de manière transparente pour l'utilisateur.

8.1.3 Syntaxe de PL/SQL

Voici maintenant une présentation plus systématique du langage PL/SQL. Elle vise à expliquer et illustrer ses principes les plus intéressants et à donner les éléments nécessaires à une expérimentation sur machine mais ne couvre cependant pas toutes ses possibilités, très étendues.

Types et variables

PL/SQL reconnaît tous les types standard de SQL, plus quelques autres dont le type `Boolean` qui peut prendre les valeurs `TRUE` ou `FALSE`. Il propose également deux constructeurs permettant de créer des types composés :

1. le constructeur `RECORD` est comparable au schéma d'une table ; il décrit un ensemble d'attributs typés et nommés ;
2. le constructeur `TABLE` correspond aux classiques tableaux unidimensionnels.

Le constructeur `RECORD` est particulièrement intéressant pour représenter un n-uplet d'une table, et donc pour définir des variables servant à stocker le résultat d'une requête SQL. On peut définir soit-même un type avec `RECORD`, avec une syntaxe très similaire à celle du `create TABLE`.

```
DECLARE
  -- Déclaration d'un nouveau type
  TYPE adresse is RECORD
   (no         integer,
    rue        varchar(40),
    ville      varchar(40),
    codePostal varchar(10)
   );
```

Mais PL/SQL offre également un mécanisme extrêmement utile consistant à dériver automatiquement un type `RECORD` en fonction d'une table ou d'un attribut d'une table. On utilise alors le nom de la table ou de l'attribut, associées respectivement au qualificateur `%ROWTYPE` ou à `%TYPE` pour désigner le type dérivé. Voici quelques exemples :

1. `Film.titre%TYPE` est le type de l'attribut `titre` de la table `Film` ;
2. `Artiste%ROWTYPE` est un type `RECORD` correspondant aux attributs de la table `Artiste`.

Le même principe de dérivation automatique d'un type s'applique également aux requêtes SQL définies dans le cadre des curseurs. Nous y reviendrons au moment de la présentation de ces derniers.

La déclaration d'une variable consiste à donner son nom, son type, à indiquer si elle peut être `NULL` et a donner éventuellement une valeur initiale. Elle est de la forme :

```
<nomVariable> <typeVariable> [NOT NULL] [:= <valeurDéfaut>]
```

Il est possible de définir également des constantes, avec la syntaxe :

```
<nomConstante> ConSTANT <typeConstante> := <valeur>
```

Toutes les déclarations de variables ou de constantes doivent être comprises dans la section `DECLARE`. Toute variable non initialisée est à `NULL`. Voici quelques exemples de déclarations. Tous les noms de variables sont systématiquement préfixés par `v_`. Ce n'est pas une obligation mais ce type de convention permet de distinguer plus facilement les variables de PL/SQL des attributs des tables dans les ordres SQL.

```
DECLARE
  -- Constantes
  v_aujourdhui  ConSTANT DATE  := SYSDATE;
  v_pi          ConSTANT NUMBER(7,5) := 3.14116;

  -- Variables scalaires
  v_compteur    integer NOT NULL := 1;
  v_nom         varchar(30);

  -- Variables pour un n-uplet de la table Film et pour le résumé
  v_film Film%ROWTYPE;
  v_resume Film.resume%TYPE;
```

Structures de contrôle

L'affectation d'une variable est effectuée par l'opérateur `:=` avec la syntaxe :

```
<nomVariable> := <expression>;
```

où `expression` est toute expression valide retournant une valeur de même type que celle de la variable. Rappelons que tous les opérateurs SQL (arithmétiques, concaténation de chaînes, manipulation de dates) et toutes les fonctions du SGBD sont utilisables en PL/SQL. Un autre manière d'affecter une variable est d'y transférer tout ou partie d'un n-uplet provenant d'une requête SQL avec la syntaxe :

```
select <nomAttribut1>, [<nomAttribut2>, ...]
into   <nomVariable1>, [<nomVariable2>, ... ]
from   [...]
```

La variable doit être du même type que l'attribut correspondant de la clause `select`, ce qui incite fortement à utiliser le type dérivé avec `%TYPE`. Dès que l'on veut transférer plusieurs valeurs d'attributs dans des variables, on a sans doute intérêt à utiliser un type dérivé `%ROWTYPE` qui limite le nombre de déclarations à effectuer. L'exemple suivant illustre l'utilisation de la clause `select ... into` associée à des types dérivés. La fonction renvoie le titre du film concaténé avec le nom du réalisateur.

```
-- Retourne une chaîne avec le titre du film et sont réalisateur

create or replace FUNCTIon TitreEtMES(v_idFilm integer) RETURN varchar is

  -- Déclaration des variables
  v_titre Film.titre%TYPE;
  v_idMES Film.idMES%TYPE;
  v_mes  Artiste%ROWTYPE;

begin
  -- Recherche du film
  select  titre, idMES
  into v_titre, v_idMES
  from Film
  where idFilm=v_idFilm;

  -- Recherche du metteur en scène
  select * into v_mes from Artiste where idArtiste = v_idMES;

  return v_titre  || ', réalisé par ' || v_mes.prenom
                  || ' ' || v_mes.nom;
end;
/
```

L'association dans la requête SQL de noms d'attributs et de noms de variables peut parfois s'avérer ambiguë d'où l'utilité d'une convention permettant de distinguer clairement ces derniers.

Les structures de test et de boucles sont tout à fait standard. La structure conditionnelle est le `if` dont la syntaxe est la suivante :

```
if <condition> then
  <instructions1>;
else
 <instruction2>;
end if;
```

Les conditions sont exprimées comme dans une clause `where` de SQL, avec notamment la possibilité de tester si une valeur est à `NULL`, des opérateurs comme `LIKE` et les connecteurs usuels `and`, `or` et `NOT`. Le `else` est optionnel, et peut éventuellement être associé à un autre `if`, selon la syntaxe généralisée suivante :

```
if <condition 1> then
  <instructions 1>;
ELSif <condition 2> then
 <instruction 2>;
  [...]
ELSif <condition n> then
 <instruction n>;
```

(suite sur la page suivante)

(suite de la page précédente)

```
else
 <instruction n+1>;
end if;
```

Il existe trois formes de boucles : `loop`, `for` et `while`. Seules les deux dernières sont présentées ici car elles suffisent à tous les besoins et sont semblables aux structures habituelles.

La boucle `while` répète un ensemble d'instructions tant qu'une condition est vérifiée. La condition est testée à chaque entrée dans la boucle. Voici la syntaxe :

```
while <condition> loop
  <instructions>;
end loop;
```

Rappelons que les expressions booléennes en SQL peuvent prendre trois valeurs : `TRUE`, `FALSE` et `UNKNOWN` quand l'évaluation de l'expression rencontre une valeur à `NULL`. Une condition est donc vérifiée quand elle prend la valeur `TRUE`, et une boucle `while` s'arrête en cas de `FALSE` ou `UNKNOWN`.

La boucle `for` permet de répéter un ensemble d'instructions pour chaque valeur d'un intervalle de nombres entiers. La syntaxe est donnée ci-dessous. Notez les deux points entre les deux bornes de l'intervalle, et la possibilité de parcourir cet intervalle de haut en bas avec l'option `REVERSE`.

```
for <variableCompteur> in [REVERSE] <min>..<max> loop
  <instructions>;
end loop;
```

Des itérations couramment utilisées en PL/SQL consistent à parcourir le résultat d'une requête SQL avec un curseur.

Structure du langage

Le code PL/SQL est structuré en *blocs*. Un bloc comprend trois sections explicitement délimitées : les déclarations, les instructions (encadrée par `begin` et `end`) et les exceptions, placées en général à la fin de la section d'instruction. On peut partout mettre des commentaires, avec deux formes possibles : soit une ligne commençant par deux tirets `--`, soit, comme en C/C++, un texte de longueur quelconque compris entre `/*` et `*/`. La structure générale d'un bloc est donc la suivante :

```
[DECLARE]
  -- Déclaration des variables, constantes, curseurs et exceptions

begin
  -- Instructions, requêtes SQL, structures de contrôle

exception
  -- Traitement des erreurs
end;
```

Le bloc est l'unité de traitement de PL/SQL. Un bloc peut être anonyme. Il commence alors par l'instruction DECLARE, et orACLE le compile et l'exécute dans la foulée au moment où il est rencontré. Le premier exemple que nous avons donné est un bloc anonyme.

Un bloc peut également être nommé (cas des procédures et fonctions) et stocké. Dans ce cas le DECLARE est remplacé par l'instruction create. Le SGBD stocke la procédure ou la fonction et l'exécute quand on l'appelle dans le cadre d'un langage de programmation. La syntaxe de création d'une procédure stockée est donnée ci-dessous.

```
create [or replace] procedure <nomProcédure>
  [(<paramètre 1>, ... <paramètre n>)] as
 [<déclarations<]
begin
 <instructions>;

 [exception
    <gestionExceptions>;
 ]
end;
```

La syntaxe des fonctions est identique, à l'exception d'un RETURN <type> précédant le as et indiquant le type de la valeur renvoyée. Procédures et fonctions prennent en entrée des *paramètres* selon la syntaxe suivante :

```
<nomParamètre> [in | out | in out] <type> [:= <valeurDéfaut>]
```

La déclaration des paramètres ressemble à celle des variables. Tous les types PL/SQL sont acceptés pour les paramètres, notamment les types dérivés avec %TYPE et %ROWTYPE, et on peut définir des valeurs par défaut. Cependant la longueur des chaînes de caractères (CHAR ou varchar) ne doit pas être précisée pour les paramètres. La principale différence avec la déclaration des variables est le mode d'utilisation des paramètres qui peut être in, out ou in out. Le mode détermine la manière dont les paramètres servent à communiquer avec le programme appelant :

1. in indique que la valeur du paramètre peut être lue mais pas être modifiée ; c'est le mode par défaut ;
2. out indique que la valeur du paramètre peut être modifée mais ne peut pas être lue ;
3. in out indique que la valeur du paramètre peut être lue et modifiée.

En d'autres termes les paramètres in permettent au programme appelant de passer des valeurs à la procédure, les paramètres out permettent à la procédure de renvoyer des valeurs au programme appelant, et les paramètres in out peuvent jouer les deux rôles. L'utilisation des paramètres out permet à une fonction ou une procédure de renvoyer plusieurs valeurs.

Gestion des erreurs

Les *exceptions* en PL/SQL peuvent être soit des erreurs renvoyées par le SGBD lui-même en cas de manipulation incorrecte des données, soit des erreurs définies par le programmeur lui-même. Le principe est que toute erreur rencontrée à l'exécution entraîne la levée} (`RAisE`) d'une exception, ce qui amène le flux de l'exécution à se dérouter vers la section `exception` du bloc courant. Cette section rassemble les actions (*exception handlers*) à effectuer pour chaque type d'exception rencontrée. Voici quelques-unes des exceptions les plus communes levées par le SGBD.

1. `inVALID_NUMBER`, indique une conversion impossible d'une chaîne de caractères vers un numérique ;
2. `inVALID_cursor`, indique une tentative d'utiliser un nom de curseur inconnu ;
3. `NO_DATA_found`, indique une requête SQL qui ne ramène aucun n-uplet ;
4. `TOO_MANY_ROWS`, indique une requête `select ... into` qui n'est pas traitée par un curseur alors qu'elle ramène plusieurs n-uplets.

Les exceptions utilisateurs doivent être définies dans la section de déclaration avec la syntaxe suivante.

```
<nomException> exception;
```

On peut ensuite lever, au cours de l'exécution d'un bloc PL/SQL, les exceptions, systèmes ou utilisateurs, avec l'instruction `RAisE`.

```
RAisE <nomException>;
```

Quand une instruction `RAisE` est rencontrée, l'exécution PL/SQL est dirigée vers la section des exceptions, et recherche si l'exception levée fait l'objet d'un traitement particulier. Cette section elle-même consiste en une liste de conditions de le forme :

```
when <nomException> then
   <traitementException>;
```

Si le nom de l'exception levée correspond à l'une des conditions de la liste, alors le traitement correspondant est exécuté. Sinon c'est la section `others` qui est utilisée. S'il n'y a pas de section gérant les exceptions, l'exception est passée au programme appelant. La procédure suivante montre quelques exemples d'exceptions.

```
-- Illustration des exceptions. La procédure prend un
-- identifiant de film, et met le titre en majuscules.
-- Les exceptions suivantes sont levées:
--   Exception système: NO_DATA_found si le film n'existe pas
--   Exception utilisateur: DEJA_FAIT si le titre
--            est déjà en majuscule

create or replace procedure TitreEnMajuscules (p_idFilm inT) as

  -- Déclaration des variables
  v_titre Film.titre%TYPE;
  deja_fait exception;
begin
```

(suite sur la page suivante)

(suite de la page précédente)

```
  -- Recherche du film. Une exception est levée si on ne trouve rien
  select titre into v_titre
  from Film where idFilm = p_idFilm;

  -- Si le titre est déjà en majuscule, on lève une autre
  -- exception
  if (v_titre = UPPER(v_titre)) then
      RAisE deja_fait;
  end if;

  -- Mise à jour du titre
   update Film set titre=UPPER(v_titre) where idFilm=p_idFilm;

  exception
    when NO_DATA_found then
     DBMS_outPUT.PUT_LinE('Ce film n''existe pas');

    when deja_fait then
     DBMS_outPUT.PUT_LinE('Le titre est déjà en majuscules');

    when others then
     DBMS_outPUT.PUT_LinE('Autre erreur...');
end;
/
```

Voici quelques exécutions de cette procédure qui montrent comment les exceptions sont levées selon le cas. On peut noter qu'orACLE considère comme une erreur le fait un ordre `select` ne ramène aucun n-uplet, et lève alors l'exception `NO_DATA_found`.

```
SQL> execute TitreEnMajuscules(900);
Le film n'existe pas

SQL> execute TitreEnMajuscules(5);

SQL> execute TitreEnMajuscules(5);
Le titre est déjà en majuscules
```

8.1.4 Quiz

8.2 S2. Les curseurs

Supports complémentaires :
— Diapositives: les curseurs
— Vidéo sur les curseurs

Comme nous l'avons indiqué précédemment, les *curseurs* constituent un mécanisme de base dans les programmes accèdant aux bases de données. Ce mécanisme repose sur l'idée de traiter *un n-uplet à la fois* dans le résultat d'une requête, ce qui permet notamment d'éviter le chargement, dans l'espace mémoire du client, d'un ensemble qui peut être très volumineux.

Le traitement d'une requête par un curseur a un impact sur le style de programmation et l'intégration avec un langage procédural, sur les techniques d'évaluation de requêtes, et sur la gestion de la concurrence d'accès. Ces derniers aspects sont traités dans d'autres chapitres. La présentation qui suit est générale pour ce qui concerne les concepts, et s'appuie sur PL/SQL pour les exemples concrets. L'avantage de PL/SQL est de proposer une syntaxe très claire et un ensemble d'options qui mettent bien en valeur les points importants.

8.2.1 Déclaration d'un curseur

Un curseur doit être déclaré dans la section `DECLARE` du bloc PL/SQL. En général on déclare également une variable dont le type est dérivé de la définition du curseur. Voici un exemple de ces deux déclarations associées :

```
-- Déclaration d'un curseur
cursor MonCurseur is
select * from Film, Artiste
where idMES = idArtiste;

-- Déclaration de la variable
v_monCurseur MonCurseur%ROWTYPE;
```

Le type de la variable, `MonCurseur%ROWTYPE`, est le type automatiquement calculé par PL/SQL pour représenter un n-uplet du résultat de la requête définie par le curseur `MonCurseur`. Ce typage permet, sans avoir besoin d'effectuer des déclarations et d'énumérer de longues listes de variables réceptrices au moment de l'exécution de la requête, de transférer très simplement chaque n-uplet du résultat dans une structure du langage procédural. Nous verrons que les choses sont beaucoup plus laborieuses avec un langage, comme le C, dont l'intégration avec SQL n'est pas du tout naturelle.

En général on utilise des curseurs *paramétrés* qui, comme leur nom l'indique, intègrent dans la requête SQL une ou plusieurs variables dont les valeurs, au moment de l'exécution, déterminent le résultat et donc l'ensemble de n-uplets à parcourir. Enfin on peut, optionnellement, déclarer l'intention de *modifier* les n-uplets traités par le curseur avec un `update`, ce qui entraîne au moment de l'exécution quelques conséquences importantes sur lesquelles nous allons revenir. La syntaxe générale d'un curseur est donc la suivante :

```
cursor <nomCurseur> [(<listeParamètres>)]
is <requête>
[for update]
```

Les paramètres sont indiqués comme pour une procédure, mais le mode doit toujours être `in` (cela n'a pas de sens de modifier le paramètre d'un curseur). Le curseur suivant effectue la même jointure que précédemment, mais les films sont sélectionnés sur l'année de parution grâce à un paramètre.

```
-- Déclaration d'un curseur paramétré
cursor MonCurseur (p_annee integer) is
 select * from Film, Artiste
 where idMES = idArtiste
 and   annee = p_annee;
```

Une déclaration complémentaire est celle des variables qui vont permettre de recevoir les n-uplets au fur et à mesure de leur parcours. Le *type dérivé* d'un curseur est obtenu avec la syntaxe `<nomCurseur>%ROWTYPE`. Il s'agit d'un type `RECORD` avec un champ par correspondant à expression de la clause `select`. Le type de chaque champ est aisément déterminé par le système. Déterminer le nom du champ est un peu plus délicat car la clause `select` peut contenir des attributs (c'est le plus courant) mais aussi des expressions construites sur ces attributs comme, par exemple, `AVG(annee)`. Il est indispensable dans ce dernier cas de donner un alias à l'expression, qui deviendra le nom du champ dans le type dérivé. Voici un exemple de cette situation :

```
-- Déclaration du curseur
cursor MonCurseur  is
 select prenom || nom as nomRéalisateur, anneeNaiss, count(*) as nbFilms
 from Film, Artiste
 where idMES = idArtiste
-- Déclaration d'une variable associée
v_realisateur MonCurseur%ROWTYPE;
```

Le type dérivé a trois champs, nommés respectivement `nomRealisateur`, `anneeNaiss` et `nbFilms`.

8.2.2 Exécution d'un curseur

Un curseur est toujours exécuté en trois phases :

1. ouverture du curseur (ordre `open`) ;
2. parcours du résultat en itérant des ordres `fetch` autant de fois que nécessaire ;
3. fermeture du curseur (`close`).

Il faut bien être conscient de la signification de ces trois phases. Au moment du `open`, le SGBD va analyser la requête, construire un plan d'exécution (un programme d'accès aux fichiers) pour calculer le résultat, et initialiser ce programme de manière à être en mesure de produire un n-uplet dès qu'un `fetch` est reçu. Ensuite, à chaque `fetch`, le n-uplet courant est envoyé par le SGBD au curseur, et le plan d'exécution se prépare à produire le n-uplet suivant.

En d'autres termes le résultat est déterminé au moment du `open`, puis exploité au fur est à mesure de l'appel des `fetch`. Quelle que soit la période sur laquelle se déroule cette exploitation (10 secondes, 1 heure ou

une journée entière), le SGBD doit assurer que les données lues par le curseur refléteront l'état de la base au moment de l'ouverture du curseur. Cela signifie notamment que les modifications effectuées par d'autres utilisateurs, ou par le programme client (c'est-à-dire celui qui exécute le curseur) lui-même, ne doivent pas être visibles au moment du parcours du résultat.

Les systèmes relationnels proposent différents niveaux d'isolation pour assurer ce type de comportement (pour en savoir plus, voir le chapitre sur la concurrence d'accès dans http://sys.bdpedia.fr). Il suffit d'imaginer ce qui se passerait si le curseur était sensible à des insertions, mises à jour ou suppressions effectuées pendant le parcours du résultat. Voici par exemple un pseudo-code montrant une situation où le parcours du curseur ne finirait jamais !

```
-- Un curseur qui s'exécute indéfiniment
open du curseur sur la table T;
while (fetch du curseur ramène un n-upletdans T) loop
  Insérer un nouveau n-uplet dans T;
end loop;
close du curseur;
```

Chaque passage dans le `where` entraîne l'insertion d'un nouveau n-uplet, et on se sortirait donc jamais de la boucle si le curseur prenait en compte ce dernier.

D'autres situations, moins caricaturales, et résultant d'actions effectuées par d'autres utilisateurs, poseraient des problèmes également. Le SGBD assure que le résultat est figé au moment du `open` puisque c'est à ce moment-là que la requête est constituée et – au moins conceptuellement – exécutée. On dit que le résultat d'un curseur est *immuable* (*immutable* en anglais).

Une solution triviale pour satisfaire cette contrainte est le calcul complet du résultat au moment du `open`, et son stockage dans une table temporaire. Cette technique présente cependant de nombreux inconvénients :

1. il faut stocker le résultat quelque part, ce qui est pénalisant s'il est volumineux ;
2. le programme client doit attendre que l'intégralité du résultat soit calculé avant d'obtenir le premièr n-uplet ;
3. si le programme client souhaite effectuer des mises à jour, il faut réserver des n-uplets qui ne seront peut-être traités que dans plusieurs minutes voire plusieurs heures.

Dire que le résultat est immuable ne signifie par forcément qu'il est calculé et matérialisé quelque part. Les chapitres consacrés à l'évaluation de requêtes et à la concurrence d'accès dans http://sys.bdpedia.fr décrivent en détail les techniques plus sophistiquées pour gérer les curseurs. Ce qu'il faut retenir ici (et partout où nous parlerons de curseur), c'est que le résultat d'une requête n'est pas forcément pré-calculé dans son intégralité mais peut être construit, utilisé puis détruit au fur et à mesure de l'itération sur les ordres `fetch`.

Ce mode d'exécution explique certaines restrictions qui semblent étranges si on n'en est pas averti. Par exemple un curseur ne fournit pas d'information sur le nombre de n-uplets du résultat, puisque ces n-uplets, parcourus un à un, ne permettent pas de savoir à l'avance combien on va en rencontrer. De même, on ne sait pas revenir en arrière dans le parcours d'un résultat puisque les n-uplets produits ne sont parfois pas conservés.

Il existe dans la norme une option `SCROLL` indiquant que l'on peut choisir d'aller en avançant ou en reculant sur l'ensemble des n-uplets. Cette option n'est disponible dans aucun système, du moins à l'heure où ces lignes sont écrites. Le `SCROLL` est problématique puisqu'il impose de conserver au SGBD le résultat complet pendant toute la durée de vie du curseur, l'utilisateur pouvant choisir de se déplacer d'avant en arrière sur

l'ensemble des n-uplets. Le SCROLL est difficilement compatible avec la technique d'exécution employés dans tous les SGBD, et qui ne permet qu'un seul parcours séquentiel sur l'ensemble du résultat.

8.2.3 Les curseurs PL/SQL

La gestion des curseurs dans PL/SQL s'appuie sur une syntaxe très simple et permet, dans un grand nombre de cas, de limiter au maximum les déclarations et instructions nécessaires. La manière la plus générale de traiter un curseur, une fois sa déclaration effectuée, et de s'appuyer sur les trois instructions `open`, `fetch` et `close` dont la syntaxe est donnée ci-dessous.

```
open <nomCurseur>[(<valeursParamètres>)];
fetch <nomCurseur> into <variableRéceptrice>;
close <nomCurseur>;
```

La (ou les) variable(s) qui suivent le `into` doivent correspondre au type d'un n-uplet du résultat. En général on utilise une variable déclarée avec le type dérivé du curseur, `<nomCurseur>%ROWTYPE`.

Une remarque importante est que les curseurs ont l'inconvénient d'une part de consommer de la mémoire du côté serveur, et d'autre part de bloquer d'autres utilisateurs si des n-uplets doivent être réservés en vue d'une mise à jour (option `for update`). Une bonne habitude consiste à effectuer le `open` le plus tard possible, et le `close` le plus tôt possible après le dernier `fetch`.

Au cours de l'accès au résultat (c'est-à-dire après le premier `fetch` et avant le `close`), on peut obtenir les informations suivantes sur le statut du curseur.

1. `<nomCurseur>%found` est un booléen qui vaut `TRUE` si le dernier `fetch` a ramené un n-uplet ;
2. `<nomCurseur>%NOTfound` est un booléen qui vaut `TRUE` si le dernier `fetch` n'a pas ramené de n-uplet ;
3. `<nomCurseur>%ROWcount` est le nombre de n-uplets parcourus jusqu'à l'état courant (en d'autres termes c'est le nombre d'appels `fetch`) ;
4. `<nomCurseur>%isopen` est un boolén qui indique si un curseur a été ouvert.

Cela étant dit, le parcours d'un curseur consiste à l'ouvrir, à effectuer une boucle en effectuant des `fetch` tant que l'on trouve des n-uplets (et qu'on souhaite continuer le traitement), enfin à fermer le curseur. Voici un exemple assez complet qui utilise un curseur paramétré pour parcourir un ensemble de films et leurs metteur en scène pour une année donnée, en afichant à chaque `fetch` le titre, le nom du metteur en scène et la liste des acteurs. Remarquez que cette liste est elle-même obtenu par un appel à la fonction PL/SQL `MesActeurs`.

```
-- Exemple d'un curseur pour rechercher les films
-- et leur metteur en scène pour une année donnée

create or replace procedure CurseurFilms (p_annee inT) as

  -- Déclaration d'un curseur paramétré
  cursor MonCurseur (v_annee integer) is
   select idFilm, titre, prenom, nom
   from Film, Artiste
   where idMES = idArtiste
```

(suite sur la page suivante)

(suite de la page précédente)

```
   and   annee = v_annee;

  -- Déclaration de la variable associée au curseur
  v_monCurseur MonCurseur%ROWTYPE;
  -- Déclaration de la variable pour la liste des acteurs
  v_mesActeurs varchar(255);
begin
  -- Ouverture du curseur
  open MonCurseur(p_annee);

  -- On prend le premier n-uplet
  fetch MonCurseur into v_monCurseur;
  -- Boucle sur les n-uplets
  while (MonCurseur%found) loop
    -- Recherche des acteurs avec la fonction MesActeurs
    v_mesActeurs := MesActeurs (v_monCurseur.idFilm);

    DBMS_outPUT.PUT_LinE('Ligne ' || MonCurseur%ROWcount ||
           ' Film:  ' || v_monCurseur.titre ||
           ', de ' || v_monCurseur.prenom || ' ' ||
           v_monCurseur.nom || ', avec ' || v_mesActeurs);
    -- Passage au n-uplet suivant
    fetch MonCurseur into v_monCurseur;
  end loop;
  -- Fermeture du curseur
  close MonCurseur;

  exception
    when others then
     DBMS_outPUT.PUT_LinE('Problème dans CurseurFilms : ' ||
                          sqlerrm);

end;
/
```

Le petit extrait d'une session sous SQL*Plus donné ci-dessous montre le résultat d'un appel à cette procédure pour l'année 1992.

```
SQL> set serveroutput on
SQL> execute CurseurFilms(1992);
Ligne 1 Film:  Impitoyable, de Clint Eastwood, avec
         Clint Eastwood, Gene Hackman, Morgan Freeman
Ligne 2 Film:  Reservoir dogs, de Quentin Tarantino, avec
          Quentin Tarantino, Harvey Keitel, Tim Roth, Chris Penn
```

La séquence des instructions `open`, `fetch` et `close` et la plus générale, notamment parce qu'elle permet de s'arrêter à tout moment en interrompant la boucle. On retrouve cette structure dans les langages de program-

mations comme C, Java et PHP. Elle a cependant l'inconvénient d'obliger à écrire deux instructions `fetch`, l'une avant l'entrée dans la boucle, l'autre à l'intérieur. PL/SQL propose une syntaxe plus concise, basée sur la boucle `for`, en tirant partie de sa forte intégration avec SQL qui permet d'inférer le type manipulé en fonction de la définition d'un curseur. Cette variante de `for` se base sur la syntaxe suivante :

```
for <variableCurseur> in <nomCurseur> loop
  <instructions;
end loop;
```

L'économie de cette construction vient du fait qu'il n'est nécessaire ni de déclarer la variable `variableCurseur`, ni d'effectuer un `open`, un `close` ou des `fetch`. Tout est fait automatiquement par PL/SQL, la variable étant définie uniquement dans le contexte de la boucle. Voici un exemple qui montre également comment traiter des mises sur les n-uplets sélectionnés.

```
-- Exemple d'un curseur effectuant des mises à jour
-- On parcourt la liste des genres, et on les met en majuscules,
-- on détruit ceux qui sont à NULL

create or replace procedure CurseurMAJ  as

  -- Déclaration du curseur
  cursor CurseurGenre is
   select * from Genre  for update;
begin
  -- Boucle for directe: pas de open, pas de close

  for v_genre in CurseurGenre loop
   if (v_genre.code is NULL) then
     delete from Genre where current of CurseurGenre;
   else
     update Genre set code=UPPER(code)
     where current of CurseurGenre;
   end if;
  end loop;
end;
/
```

Notez que la variable `v_genre` n'est pas déclarée explicitement. Le curseur est défini avec une clause `for update` qui indique au SGBD qu'une mise à jour peut être effectuée sur chaque n-uplet. Dans ce cas – et dans ce cas seulement – il est possible de faire référence au n-uplet courant, au sein de la boucle `for`, avec la syntaxe `where current of <nomCurseur>`.

Si on n'a pas utilisé la clause `for update`, il est possible de modifier (ou détruire) le n-uplet courant, mais en indiquant dans le `where` de la clause `update` la valeur de la clé. Outre la syntaxe légèrement moins concise, cette désynchronisation entre la lecture par le curseur, et la modification par SQL, entraîne des risques d'incohérence (mise à jour par un autre utilisateur entre le `open` et le `fetch`) qui sont développés dans le chapitre consacré à la concurrence d'accès (http://sys.bdpedia.fr).

Il existe une syntaxe encore plus simple pour parcourir un curseur en PL/SQL. Elle consiste à ne pas déclarer

explicitement de curseur, mais à placer la requête SQL directement dans la boucle `for`, comme par exemple :

```
for v_genre in (select * from Genre) loop
  <instructions;
end loop;
```

Signalons pour conclure que PL/SQL traite *toutes* les requêtes SQL par des curseurs, que ce soit des ordres `update`, `insert`, `delete` ou des requêtes `select` ne ramenant qu'une seule ligne. Ces curseurs sont « implicites » car non déclarés par le programmeur, et tous portent le même nom conventionnel, `SQL`. Concrètement, cela signifie que les valeurs suivantes sont définies après une mise à jour par `update`, `insert`, `delete` :

1. `SQL%found` vaut `TRUE` si la mise à jour a affecté au moins un n-uplet ;
2. `SQL%NOTfound` vaut `TRUE` si la mise à jour n'a affecté aucun n-uplet ;
3. `SQL%ROWcount` est le nombre de n-uplets affecté par la mise à jour ;
4. `SQL%isopen` renvoie systématiquement `FALSE` puisque les trois phases (ouverture, parcours et fermeture) sont effectuées solidairement.

Le cas du `select` est un peu différent : une exception est toujours levée quand une recherche sans curseur ne ramène pas de n-uplet (exception `NO_DATA_found`) ou en ramène plusieurs (exception `TOO_MANY_ROWS`). Il faut donc être prêt à traiter ces exceptions pour ce type de requête. Par exemple, la recherche :

```
select * into v_film
from Film
where titre LIKE 'V%';
```

devrait être traitée par un curseur car il y n'y a pas de raison qu'elle ramène un seul n-uplet.

8.2.4 Quiz

8.3 S3. Les déclencheurs

Supports complémentaires :
— Diapositives: les déclencheurs
— Vidéo sur les déclencheurs

Le mécanisme de *triggers* (que l'on peut traduire par « déclencheur » ou « réflexe ») est implanté dans les SGBD depuis de nombreuses années, et a été normalisé par SQL99. Un *trigger* est simplement une procédure stockée dont la particularité principale est de se déclencher automatiquement sur certains événements mise à jour *spécifiés par le créateur du trigger*.

On peut considérer les *triggers* comme une extension du système de contraintes proposé par la clause `CHECK` : à la différence de cette dernière, l'événement déclencheur est explicitement indiqué, et l'action n'est pas limitée à la simple alternative acceptation/rejet. Les possibilités offertes par les *triggers* sont très intéressantes. Citons :

— la gestion des redondances ; l'enregistrement automatique de certains évènenements (*auditing*) ;
— la spécification de contraintes complexes liées à l'évolution des données (exemple : le prix d'une séance ne peut qu'augmenter) ;

— toute règle liée à l'environnement d'exécution (restrictions sur les horaires, les utilisateurs, etc.).

Les *triggers* sont discutés dans ce qui suit de manière générale, et illustrés par des exemples orACLE. Il faut mentionner que la syntaxe de déclaration des *triggers* est suivie par la plupart des SGBD, les principales variantes se situant au niveau du langage permettant d'implanter la partie procédurale. Dans ce qui suit, ce langage sera bien entendu PL/SQL.

8.3.1 Principes des *triggers*

Le modèle d'exécution des *triggers* est basé sur la séquence *Evénement-Condition-Action* (ECA) que l'on peut décrire ainsi :

— un *trigger* est déclenché par un *événement*, spécifié par le programmeur, qui est en général une insertion, destruction ou modification sur une table ;
— la première action d'un *trigger* est de tester une *condition* : si cette condition ne s'évalue pas à `TRUE`, l'exécution s'arrête ;
— enfin *l'action* proprement dite peut consister en toute ensemble d'opérations sur la base de données, effectuée si nécessaire à l'aide du langage procédural supporté par le SGBD.

Une caractéristique importante de cette procédure (action) est de pouvoir manipuler simultanément les valeurs ancienne et nouvelle de la donnée modifiée, ce qui permet de faire des tests sur l'évolution de la base.

Parmi les autres caractéristiques importantes, citons les deux suivantes. Tout d'abord un *trigger* peut être exécuté au choix une fois pour un seul ordre SQL, ou à chaque n-uplet concerné par cet ordre. Ensuite l'action déclenchée peut intervenir *avant* l'événement, ou *après*.

L'utilisation des *triggers* permet de rendre une base de données *dynamique* : une opération sur la base peut en déclencher d'autres, qui elles-mêmes peuvent entraîner en cascade d'autres réflexes. Ce mécanisme n'est pas sans danger à cause des risques de boucle infinie.

Prenons l'exemple suivant : on souhaite conserver au niveau de la table `Cinéma` le nombre total de places (soit la somme des capacités des salles). Il s'agit en principe d'une redondance à éviter, mais que l'on peut gérer avec les *triggers*. On peut en effet implanter un *trigger* au niveau `Salle` qui, pour toute mise à jour, va aller modifier la donnée au niveau `Cinéma`.

Maintenant il est facile d'imaginer une situation où on se retrouve avec des *triggers* en cascade. Prenons le cas d'une table `Ville (nom, capacité)` donnant le nombre de places de cinéma dans la ville.

Maintenant, supposons que la ville gère l'heure de la première séance d'une salle : on aboutit à un cycle infini !

8.3.2 Syntaxe

La syntaxe générale de création d'un *trigger* est donnée ci-dessous.

```
 create [or replace] trigger <nomTrigger>
  {before | after}
  {delete | insert | update [of column, [, column] ...]}
     [ or {delete | insert | update [of column, [, column] ...]}] ...
  on <nomTable> [for each row]
[when <condition]
   <blocPLSQL>
```

On peut distinguer trois parties dans cette construction syntaxique. La partie *événement* est spécifiée après `before` ou `after`, la partie *condition* après `when` et la partie *action* correspond au bloc PL/SQL. Voici quelques explications complémentaires sur ces trois parties.

— « Evénement », peut être ` before`` ou `after`, suivi de `delete`, `update` ou `insert` séparés par des `or`.

— « Condition », `for each row` est optionnel. En son absence le *trigger* est déclenché une fois pour toute requête modifiant la table, *et ce sans condition*.
Sinon <condition> est toute condition booléenne SQL. De plus on peut rérférencer les anciennes et nouvelles valeurs du tuple courant avec la syntaxe `new.attribut` et `old.attribut` respectivement.

— « Action » est une procédure qui peut être implantée, sous Oracle, avec le langage PL/SQL. Elle peut contenir des ordres SQL *mais pas de mise à jour de la table courante*.
Les anciennes et nouvelles valeurs du tuple courant sont référencées par `:new.attr` et `:old.attr`. Il est possible de modifier `new` et `old`. Par exemple `:new.prix=500;` forcera l'attribut `prix` à 500 dans un `before` *trigger*.

La disponibilité de `new` et `old` dépend du contexte. Par exemple `new` est à `NULL` dans un *trigger* déclenché par `delete`.

8.3.3 Quelques exemples

Voici tout d'abord un exemple de *trigger* qui maintient la capacité d'un cinéma à chaque mise à jour sur la table `Salle`.

```
create trigger CumulCapacite
after update on Salle
for each row
when (new.capacite != old.capacite)
begin
  update Cinema
  set capacite = capacite - :old.capacite  + :new.capacite
  where  nom = :new.nomCinema;
end;
```

Pour garantir la validité du cumul, il faudrait créer des *triggers* sur les événements `update` et `insert`. Une solution plus concise (mais plus coûteuse) est de recalculer systématiquement le cumul : dans ce cas on peut utiliser un *trigger* qui se déclenche globalement pour la requête :

```
create trigger CumulCapaciteGlobal
after update or insert or delete on Salle
begin
  update Cinema C
  set capacite = (select sum (capacite)
                  from   Salle S
                  where  C.nom = S.nomCinema);
end;
```

8.3.4 Quiz

8.4 Atelier : JDBC (optionnel)

Important : Cet atelier propose une découverte de l'interface normalisée JDBC d'accès à une base relationnelle. C'est du Java : si vous ne connaissez (un peu) pas le langage et si vous ne souhaitez pas l'apprendre, contentez-vous de lire.

Ceux qui sont éauipés sont encouragés à metre en pratique ce qui suit et à compléter les exercices proposés.

Supports complémentaires :

- Le driver MySQL Vous pouvez récupérer une version plus récentes sur le site d'Oracle ou de MySQL. Pour les *drivers* des autres SGBD, à vous de trouver, ils sont tous en accès libre.
- Le programme de lecture des voyageurs

JDBC (acronyme qui signifie probablement « Java Database Connectivity » par analogie avec ODBC), est une API (*Application Programming Interface* = description d'une interface fonctionnelle) Java qui permet de se connecter à une base de données, d'interroger cette base afin d'en extraire des données, ou effectuer des mises à jour.

JDBC est complètement indépendant de tout SGBD : la même application peut être utilisée pour accéder à une base Oracle, Postgres, MySQL, etc. Conséquences : pas besoin d'apprendre une nouvelle API quand on change de SGBD, et réutilisation totale du code.

La présentation qui suit ne couvre pas tous les aspects de JDBC. Vous trouverez de très nombreux tutoriels surt le Web, de bonne qualité dans l'ensemble. Dans la suite de ce texte vous trouverez une description des principes de JDBC, et une introduction à ses fonctionnalités, essentiellement basée sur des exemples simples.

8.4.1 Principes de JDBC

L'utilisation de JDBC se fait dans le cadre d'une application Java que nous appellerons *le client*. L'application communique avec le serveur du SGBD via le réseau. Je vous renvoie à la Fig. 8.1 et aux explications qui l'accompagnent.

Le dialogue avec la base distante se fait par l'intermédiaire d'une *connexion*, qui elle-même utilise les services d'un *driver*. Chaque SGBD propose son *driver* : tous sont conformes à l'API JDBC, et chacun a la spécificité de savoir dialoguer avec le serveur dans son protocole particulier. Pour se connecter a MySQL il faut le *driver* de MySQL, pour Postgres le *driver* de Postgres et ainsi de suite. Ce n'est pas en contradiction avec la généricité du code Java. Tous les drivers ont la même interface et s'utilisent de la même façon. Pour passer d'un SGBD à un autre, notre code reste le même, il suffit d'utiliser de *driver* adapté.

Quand une requête doit être exécutée, elle le fait par l'intermédiaire d'une *connexion*. Une connexion est un objet Java de la classe *Connection* qui est chargé de dialoguer avec une base de données. Pour créer un objet `Connection`, il faut disposer d'un droit d'accès (c'est une différence avec nos exemple PL/SQL : pour exécuter sur PL/SQL,il faut *déjà* être connecté au serveur).

Dans ce qui suit, je suppose que le serveur est MySQL, en écoute sur le port 3306 de la machine `localhost` (la même que celle où s'exécute l'application donc). Le compte d'accès est le suivant :

```
grant all privileges on Voyageurs.* to moi identified by 'motdepasse'
```

Le *driver* est donc celui de MySQL. Vous vous faciliter les choses j'en ai mis un en téléchargement ci=dessus. Le fichier *jar* doit faire partie de ceux mentionnés par votre varible `CLASSPATH`. Je n'en dis pas plus : vous trouverez beaucoup de ressources sur le Web pour configurer votre environnement si vous avez des difficultés.

Note : Si vous avez Postgres ou n'importe quel autre SGBD, le code doit fonctionner aussi, il faut juste installer le *driver* correspondant et modifier l'URL de connexion, voir ci-dessous.

8.4.2 Premier exemple

Voici un premier programme JDBC Il se connecte à la base, recherche l'ensemble des voyageurs, et affiche les nom et prénom à l'écran.

```
// Import de tous les packages JDBC
import java.sql.*;

// Une classe qui affiche les noms des voyageurs
class ListeVoyageurs {
   // Méthode "main", celle qui s'exécute quand on lance le programme
   public static void main(String args[]) throws SQLException {
      // Paramètres de connexion
      String url = "jdbc:mysql://localhost:3306/Voyageurs";
      String utilisateur = "moi";
      String motdepasse = "motdepasse";

      try {
         // Connection à la base
         Connection conn = DriverManager.getConnection(url, utilisateur,␣
↪motdepasse);

         // Création d'un Statement,autrement dit un curseur
         Statement stmt = conn.createStatement();
         // Exécution de la requête qui ramène les voyageurs
         ResultSet rset = stmt.executeQuery("select * from Voyageur");

         // Affichage du résultat = parcours du curseur
         while (rset.next()) {
            System.out.println(rset.getString("prénom") + " " + rset.getString(
↪"nom"));
         }
         // On libère proprement les ressources
```

(suite sur la page suivante)

(suite de la page précédente)

```
            rset.close();
            stmt.close();
        } catch (SQLException e) {
            // Oups, quelque chose s'est mal passé
            System.out.println("Problème quelque part !!!");
            System.out.println(e.getMessage());
        }
    }
}
```

Les commentaires dans le programme vous disent à peu près tout ce qu'il faut savoir. Notez

- les paramètres de connexion : l'URL est spécifique à MySQL, elle indique l'hôte (`localhost`), le port (3306) et la base.
- l'interrogation se fait classiquement par curseur, avec un `open` (`execute` en JDBC) et une série de `next()`. Enfin, on libère les ressources avec `close()` (qu'il faut appeler deux fois, ne me demandez pas pourquoi).
- les nuplets ramenés par `next()` sont accessibles via l'interface de la classe `ResultSet` ; pour accéder aux champs, il faut connaître leur nom et leur type (pour choisir entre `getString()`, `getInt()`, etc..

Voilà. Premier exercice : récupérer le code (lien en début de section) et compilez-le

```
javac ListeVoyageurs
```

Vous pouvez alors l'exécuter

```
java ListeVoyageurs
```

Et vous devriez obtenir la liste des voyageurs. Sinon, il faut chercher ce qui cloche.

8.4.3 Exercices

Voici une suggestion d'exercices pour améliorer notre premier exemple et découvrir JDBC.

8.4.4 Requêtes préparées

La classe `PreparedStatement` permet de soumettre des requêtes dites « préparées » dans lesquelles on peut introduire des paramètres.

Etudiez cette classe (faites une requête Google pour trouver une description sur le Web) et écrivez une version du programme de lecture qui prend en paramètre la région des voyageurs.

Pour tester les mises à jour, vous pouvez également écrire un programme qui insère un nouveau voyageur. Pour bien faire, il faudrait

- vérifier que le voyageur n'existe pas, avant d'effectuer l'insertion
- trouver une valeur d'identifiant non utilisée (astuce : cherchez l'identifiant maximal, et ajoutez 1)

Note : Idéalement, les paramètres seraient fournis au programme au moment de l'appel.

8.4.5 L'interface `ResultSet`

Avec `ResultSet` vous pouvez obtenir des métadonnées sur le résultat d'une requête : le nombre de nuplets, et les noms et types des attributs.

Modifiez le programme pour afficher en entête les noms des champs. En utilisant des tabulations vous devriez pouvoir obtenir un joli affichage, bien aligné.

Vous devriez finalement être capable d'obtenir un programme auquel on passe n'importe quelle requête SQL, qui l'exécute et qui affiche le résultat avec les noms des attributs en entête. Autrement dit l'équivalent de l'interface que vous avez sans doute utilisée pour tester vos requêtes SQL. À vous de jouer.

CHAPITRE 9

Transactions

Quand on développe un programme *P* accédant à une base de données, on effectue en général plus ou moins explicitement deux hypothèses :

— *P* s'exécutera *indépendamment* de tout autre programme ou utilisateur ;
— l'exécution de *P* se déroulera toujours intégralement.

Il est clair que ces deux hypothèses ne se vérifient pas toujours. D'une part les bases de données constituent des ressources accessibles *simultanément* à plusieurs utilisateurs qui peuvent y rechercher, créer, modifier ou détruire des informations : les accès simultanés à une même ressource sont dits *concurrents*, et l'absence de contrôle de cette concurrence peut entraîner de graves problèmes de cohérence dus aux interactions des opérations effectuées par les différents utilisateurs. D'autre part on peut envisager beaucoup de raisons pour qu'un programme ne s'exécute pas jusqu'à son terme. Citons par exemple :

— l'arrêt du serveur de données ;
— une erreur de programmation entrainant l'arrêt de l'application ;
— la violation d'une contrainte amenant le système à rejeter les opérations demandées ;
— une annulation décidée par l'utilisateur.

Une interruption de l'exécution peut laisser la base dans un état transitoire *incohérent*, ce qui nécessite une opération de réparation consistant à ramener la base au dernier état cohérent connu avant l'interruption. Les SGBD relationnels assurent, par des mécanismes complexes, un partage concurrent des données et une gestion des interruptions qui permettent d'assurer à l'utilisateur que les deux hypothèses adoptées intuitivement sont satisfaites, à savoir :

— son programme se comporte, au moment où il s'exécute, *comme s'il* était seul à accéder à la base de données ;
— en cas d'interruption intempestive, les mises à jour effectuées depuis le dernier état cohérent seront annulées par le système.

On désigne respectivement par les termes de *contrôle de concurrence* et de *reprise sur panne* l'ensemble des techniques assurant ce comportement. En théorie le programmeur peut s'appuyer sur ces techniques, intégrées au système, et n'a donc pas à se soucier des interactions avec les autres utilisateurs. En pratique les choses ne sont pas si simples, et le contrôle de concurrence a pour contreparties certaines conséquences

qu'il est souvent important de prendre en compte dans l'écriture des applications. En voici la liste, chacune étant développée dans le reste de ce chapitre :

— **Définition des points de sauvegardes**. La reprise sur panne garantit le retour au dernier état cohérent de la base précédant l'interruption, mais c'est au programmeur de définir ces points de cohérence (ou *points de sauvegarde*) dans le code des programmes.
— **Blocages des autres utilisateurs**. Le contrôle de concurrence s'appuie sur le verrouillage de certaines resources (tables blocs, n-uplets), ce qui peut bloquer temporairement d'autres utilisateurs. Dans certains cas des *interblocages* peuvent même apparaître, amenant le système à rejeter l'exécution d'un des programmes en cause.
— **Choix d'un niveau d'isolation**. Une isolation totale des programmes garantit la cohérence, mais entraîne une dégradation des performances due aux verrouillages et aux contrôles appliqués par le SGBD. Or, dans beaucoup de cas, le verrouillage/contrôle est trop strict et place en attente des programmes dont l'exécution ne met pas en danger la cohérence de la base. Le programmeur peut alors choisir d'obtenir plus de concurrence (autrement dit, plus de *fluidité* dans les exécutions concurrentes), en demandant au système un niveau d'isolation moins strict, et en prenant éventuellement lui-même en charge le verrouillage des ressources critiques.

Ce chapitre est consacré à la concurrence d'accès, vue par le programmeur d'application. Il ne traite pas, ou très superficiellement, des algorithmes implantés par les SGBD. L'objectif est de prendre conscience des principales techniques nécessaires à la préservation de la cohérence dans un système multi-utilisateurs, et d'évaluer leur impact en pratique sur la réalisation d'applications bases de données. La gestion de la concurrence, du point de vue de l'utilisateur, se ramène en fait à la recherche du bon compromis entre deux solutions extrêmes :

— une cohérence maximale impliquant un risque d'interblocage relativement élevé ;
— ou une fluidité concurrentielle totale au prix de risques importants pour l'intégrité de la base.

Ce compromis dépend de l'application et de son contexte (niveau de risque acceptable *vs* niveau de performance souhaité) et relève donc du choix du concepteur de l'application. Mais pour que ce choix existe, et puisse être fait de manière éclairée, encore faut-il être conscient des risques et des conséquences d'une concurrence mal gérée. Ces conséquences sont insidieuses, souvent erratiques, et il est bien difficile d'imputer au défaut de concurrence des comportements que l'on a bien du mal à interpréter. Tout ce qui suit vise à vous éviter ce fort désagrément.

Le chapitre débute par une définition de la notion de *transaction*, et montre ensuite, sur différents exemples, les problèmes qui peuvent survenir. Pour finir nous présentons les niveaux d'isolation définis par la norme SQL.

9.1 S1 : Transactions

Supports complémentaires :

— Diapositives: la notion de transaction
— Fichier de commandes pour tester les transactions sous MySQL
— Vidéo sur la notion de transaction

Une *transaction* est une séquence d'opérations de lecture ou de mise à jour sur une base de données, se terminant par l'une des deux instructions suivantes :

— `commit`, indiquant la validation de toutes les opérations effectuées par la transaction ;

— `rollback` indiquant l'annulation de toutes les opérations effectuées par la transaction.

Une transaction constitue donc, pour le SGBD, une unité d'exécution. Toutes les opérations de la transaction doivent être validées ou annulées solidairement.

9.1.1 Notions de base

On utilise toujours le terme de transaction, au sens défini ci-dessus, de préférence à « programme », « procédure » ou « fonction », termes à la fois inappropriés et quelque peu ambigus. « Programme » peut en effet désigner, selon le contexte, la spécification avec un langage de programmation, ou l'exécution sous la forme d'un processus client communiquant avec le (programme serveur du) SGBD. C'est toujours la seconde acception qui s'applique pour le contrôle de concurrence. De plus, l'exécution d'un programme (un *processus*) consiste en une suite d'ordres SQL adressés au SGBD, cette suite pouvant être découpée en une ou plusieurs transactions en fonction des ordres `commit` ou `rollback` qui s'y trouvent. La première transaction débute avec le premier ordre SQL exécuté ; toutes les autres débutent après le `commit` ou le `rollback` de la transaction précédente.

Note : Il est aussi possible d'indiquer explicitement le début d'une transaction avec la commande `START TRANSACTION`.

Dans tout ce chapitre nous allons prendre l'exemple de transactions consistant à réserver des places de spectacle pour un client. On suppose que la base contient les deux tables suivantes :

```
create table Client (id_client INTEGER NOT NULL,
                     nom VARCHAR(255) NOT NULL,
                     nb_places_reservees INTEGER NOT NULL,
                     solde INTEGER NOT NULL,
                     primary key (id_client));
create table Spectacle (id_spectacle INTEGER NOT NULL,
                        nb_places_offertes INTEGER NOT NULL,
                        nb_places_libres INTEGER NOT NULL,
                        tarif DECIMAL(10,2) NOT NULL,
                        primary key (id_spectacle));
```

Chaque transaction s'effectue pour un client, un spectacle et un nombre de places à réserver. Elle consiste à vérifier qu'il reste suffisamment de places libres. Si c'est le cas elle augmente le nombre de places réservées par le client, et elle diminue le nombre de places libres pour le spectacle. On peut la coder en n'importe quel langage. Voici, pour être concret (et concis), la version PL/SQL.

```
/* Un programme de reservation  */

create or replace procedure Reservation (v_id_client INTEGER,
                                         v_id_spectacle INTEGER,
                                          nb_places INT) AS
-- Déclaration des variables
v_client Client%ROWTYPE;
v_spectacle Spectacle%ROWTYPE;
```

(suite sur la page suivante)

(suite de la page précédente)

```
v_places_libres INTEGER;
v_places_reservees INTEGER;
begin
-- On recherche le spectacle
select * into v_spectacle
from Spectacle where id_spectacle=v_id_spectacle;

-- S'il reste assez de places: on effectue la reservation
if (v_spectacle.nb_places_libres >= nb_places)
then
  -- On recherche le client
  select * into v_client from Client where id_client=v_id_client;

  -- Calcul du transfert
  v_places_libres := v_spectacle.nb_places_libres - nb_places;
  v_places_reservees := v_client.nb_places_reservees + nb_places;

  -- On diminue le nombre de places libres
  update Spectacle set nb_places_libres = v_places_libres
     where id_spectacle=v_id_spectacle;

  -- On augmente le nombre de places reervees par le client
  update Client set nb_places_reservees=v_places_reservees
    where id_client = v_id_client;

  -- Validation
  commit;
ELSE
 rollback;
end if;
end;
/
```

Chaque *exécution* de ce code correspondra à une transaction. La première remarque, essentielle pour l'étude et la compréhension du contrôle de concurrence, est que l'exécution d'une procédure de ce type correspond à des échanges entre deux processus distincts : le processus *client* qui exécute la procédure, et le processus *serveur* du SGBD qui se charge de satisfaire les requêtes SQL. On prend toujours l'hypothèse que les zones mémoires des deux processus sont distinctes et étanches. Autrement dit le processus client ne peut accéder aux données que par l'intermédiaire du serveur, et le processus serveur, de son côté, est totalement ignorant de l'utilisation des données transmises au processus client (Fig. 9.1).

Il s'ensuit que non seulement le langage utilisé pour coder les transactions est totalement indifférent, mais que les variables, les interactions avec un utilisateur ou les structures de programmation (tests et boucles) du processus client sont transparentes pour le programme serveur. Ce dernier ne connaît que la séquence des instructions qui lui sont explicitement destinées, autrement dit les ordres de lecture ou d'écriture, les `commit` et les `rollback`.

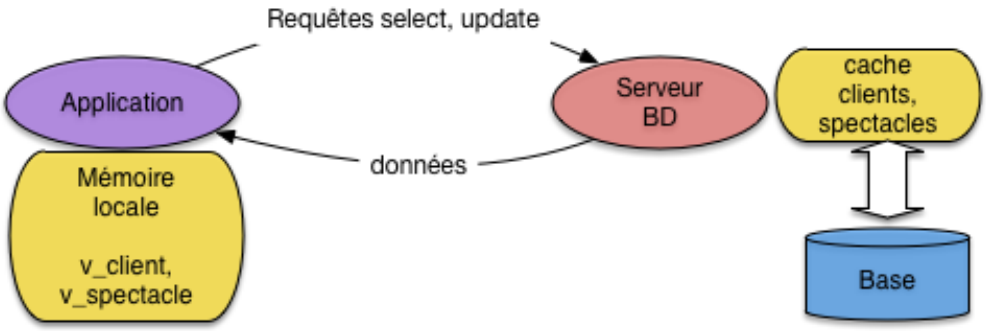

Fig. 9.1 – Le processus client et le processus serveur pendant une transaction

D'autre part, les remarques suivantes, assez triviales, méritent cependant d'être mentionnées :

- deux exécutions de la procédure ci-dessus peuvent entraîner deux transactions différentes, dans le cas par exemple où le test effectué sur le nombre de places libres est positif pour l'une est négatif pour l'autre ;
- un processus peut exécuter répétitivement une procédure –par exemple pour effectuer plusieurs réservations– ce qui déclenche des transactions *en série* (retenez le terme, il est important) ;
- deux processus distincts peuvent exécuter indépendamment la même procédure, avec des paramètres qui peuvent être identiques ou non.

On fait toujours l'hypothèse que deux processus ne communiquent jamais entre eux. *En résumé, une transaction est une séquence d'instructions de lecture ou de mise à jour transmise par un processus client au serveur du SGBD, se concluant par* `commit` ou `rollback`.

9.1.2 Exécutions concurrentes

Pour chaque processus il ne peut y avoir qu'une seule transaction en cours à un moment donné, mais plusieurs processus peuvent effectuer simultanément des transactions. C'est même le cas général pour une base de données à laquelle accèdent simultanément plusieurs applications. Si elles manipulent les *mêmes* données, on peut aboutir à un entrelacement des lectures et écritures par le serveur, potentiellement générateur d'anomalies (Fig. 9.2).

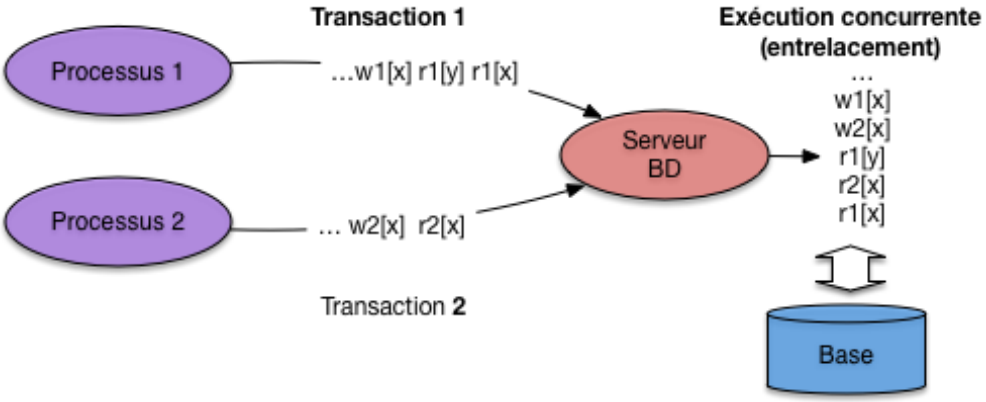

Fig. 9.2 – Exécution concurrentes engendrant un entrelacement

Chaque processus est identifié par un numéro unique qui est soumis au SGBD avec chaque ordre SQL effectué par ce processus (c'est l'identifiant de session, obtenu au moment de la connexion). Voici donc comment on représentera une transaction du processus numéro 1 exécutant la procédure de réservation.

$$read_1(s); read_1(c); write_1(s); write_1(c); C_1$$

Les symboles *c* et *s* désignent les nuplets –ici un spectacle *s* et un client *c*– lus ou mis à jour par les opérations, tandis que le symbole *C* désigne un `commit` (on utilisera bien entendu *R* pour le `rollback`). Dans tout ce chapitre on supposera, sauf exception explicitement mentionnée, que l'unité d'accès à la base est le nuplet (une ligne dans une table), et que tout verrouillage s'applique à ce niveau.

Voici une autre transaction, effectuée par le processus numéro 2, pour la même procédure.

$$read_2(s');$$

On a donc lu le spectacle *s"*, et constaté qu'il n'y a plus assez de places libres. Enfin le dernier exemple est celui d'une réservation effectuée par un troisième processus.

$$read_3(s); read_3(c'); write_3(s); write_3(c); C_3$$

Le client *c"* réserve donc ici des places pour le spectacle *s*.

Les trois processus peuvent s'exécuter au même moment, ce qui revient à soumettre à peu près simultanément les opérations au SGBD. Ce dernier pourrait choisir d'effectuer les transactions *en série*, en commençant par exemple par le processus 1, puis en passant au processus 2, enfin au processus 3. Cette stratégie a l'avantage de garantir de manière triviale la vérification de l'hypothèse d'isolation des exécutions, mais elle est potentiellement très pénalisante puisqu'une longue transaction pourrait metre en attente pour un temps indéterminé de nombreuses petites transactions.

C'est d'autant plus injustifié que, le plus souvent, l'entrelacement des opérations est sans danger, et qu'il est possible de contrôler les cas où il pourrait poser des problèmes. Tous les SGBD autorisent donc des *exécutions concurrentes* dans lequelles les opérations s'effectuent alternativement pour des processus différents. Voici un exemple d'exécution concurrente pour les trois transactions précédentes, dans lequel on a abrégé *read* et *write* respectivement par *r* et *w*.

$$r_1(s); r_3(s); r_1(c); r_2(s'); r_3(c'); w_3(s); w_1(s); w_1(c); w_3(c); C_1; C_3$$

Dans un premier temps on peut supposer que l'ordre des opérations dans une exécution concurrente est l'ordre de transmission de ces opérations au système. Comme nous allons le voir sur plusieurs exemples, cette absence de contrôle mène à de nombreuses anomalies qu'il faut absolument éviter. Le SGBD (ou, plus précisément, le module chargé du contrôle de concurrence) effectue donc un ré-ordonnancement si cela s'impose. Cependant, l'entrelacement des opérations ou leur ré-ordonnancement ne signifie *en aucun cas* que l'ordre des opérations internes à une transaction T_i peut être modifié. En « effaçant » d'une exécution concurrente toutes les opérations pour les transactions $T_j, i \neq j$ on doit retrouver exactement les opérations de T_i, dans l'ordre où elles ont été soumises au système. Ce dernier ne change *jamais* cet ordre car cela reviendrait à transformer le programme en cours d'exécution.

La Fig. 9.3 montre une première ébauche des composants intervenant dans le contrôle de concurrence. On y retrouve les trois processus précédents, chacun soumettant des instructions au serveur. Le processus 1 soumet

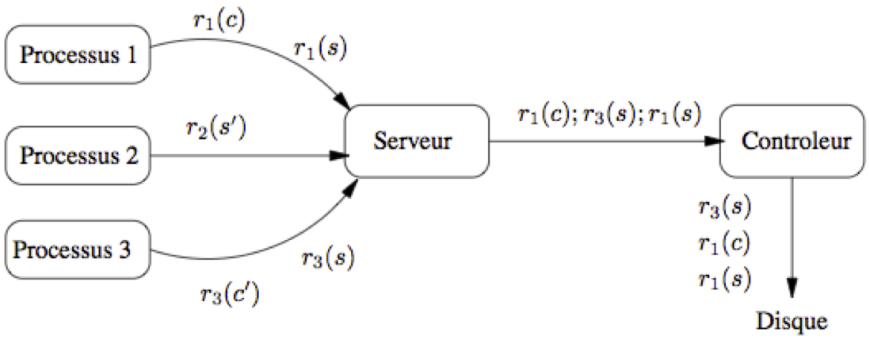

Fig. 9.3 – Processus soumettant des transactions

par exemple $r_1(s)$, puis $r_1(c)$. Le serveur transmet les instructions, dans l'ordre d'arrivée (ici, d'abord $r_1(s)$, puis $r_3(s)$, puis $r_1(c)$), à un module spécialisé, le *contrôleur* qui, lui, peut réordonnancer l'exécution s'il estime que la cohérence en dépend. Sur l'exemple de la Fig. 9.3, le contrôleur exécute, dans l'ordre $r_1(s)$, $r_1(c)$ puis $r_3(s)$.

9.1.3 Propriétés ACID des transactions

Les SGBD garantissent que l'exécution des transactions satisfait un ensemble de bonnes propriétés que l'on résume commodément par l'acronyme ACID (Atomicité, Cohérence, Isolation, Durabilité).

Isolation

L'isolation est la propriété qui garantit que l'exécution d'une transaction *semble* totalement indépendante des autres transactions. Le terme « semble » est bien entendu relatif au fait que, comme nous l'avons vu ci-dessus, une transaction s'exécute en fait en concurrence avec d'autres. Du point de vue de l'utilisateur, tout se passe donc comme si son programme, pendant la période de temps où il accède au SGBD, était seul à disposer des ressources du serveur de données.

Le niveau d'isolation totale, telle que défini ci-dessus, est dit *sérialisable* puisqu'il est équivalent du point de vue du résultat obtenu, à une exécution *en série* des transactions. C'est une propriété forte, dont l'inconvénient est d'impliquer un contrôle strict du SGBD qui risque de pénaliser fortement les autres utilisateurs. Les systèmes proposent en fait plusieurs niveaux d'isolation dont chacun représente un compromis entre la sérialisabilité, totalement saine mais pénalisante, et une isolation partielle entraînant moins de blocages mais plus de risques d'interactions perturbatrices.

Le choix du bon niveau d'isolation, pour une transaction donnée, est de la responsabilité du programmeur et implique une bonne compréhension des dangers courus et des options proposées par les SGBD. Le présent chapitre est essentiellement consacré à donner les informations nécessaires à ce choix.

Garantie de la commande `commit` (durabilité)

L'exécution d'un `commit` rend permanentes toutes les mises à jour de la base effectuées durant la transaction. Le système garantit que toute interruption du système survenant après le `commit` ne remettra pas en cause ces mises à jour. Cela signifie également que tout `commit` d'une transaction T rend impossible l'annulation de cette même transaction avec `rollback`. Les anciennes données sont perdues, et il n'est pas possible de revenir en arrière.

Le `commit` a également pour effet de lever tous les verrous mis en place durant la transaction pour prévenir les interactions avec d'autres transactions. Un des effets du `commit` est donc de « libérer » les éventuelles ressources bloquées par la transaction validée. Une bonne pratique, quand la nature de la transaction le permet, est donc d'effectuer les opérations potentiellement bloquantes le plus tard possible, juste avant le `commit` ce qui diminue d'autant la période pendant laquelle les données en concurrence sont verrouillées.

Garantie de la commande `rollback` (atomicité)

Le `rollback` annule toutes les modifications de la base effectuées pendant la transaction. Il relâche également tous les verrous posés sur les données pendant la transaction par le système, et libère donc les éventuels autres processus en attente de ces données.

Un `rollback` peut être déclenché explicitement par l'utilisateur, ou effectué par le système au moment d'une reprise sur panne ou de tout autre problème empêchant la poursuite normale de la transaction. Dans tout les cas l'état des données modifiées par la transaction revient, après le `rollback`, à ce qu'il était au début de la transaction. Cette commande garantit donc *l'atomicité* des transactions, puisqu'une transaction est soit effectuée totalement (donc jusqu'au `commit` qui la conclut) soit annulée totalement (par un `rollback` du système ou de l'utilisateur).

L'annulation par `rollback` rend évidemment impossible toute validation de la transaction : les mises à jour sont perdues et doivent être resoumises.

Cohérence des transactions

Le maintien de la cohérence peut relever aussi bien du système que de l'utilisateur selon l'interprétation du concept de « cohérence ».

Pour le système, la cohérence d'une base est définie par les contraintes associées au schéma. Ces contraintes sont notamment :

- les contraintes de clé primaire (clause `primary key`) ;
- l'intégrité référentielle (clause `foreign key`) ;
- les contraintes `check` ;
- les contraintes implantées par *triggers*.

Toute violation de ces contraintes entraîne non seulement le rejet de la commande SQL fautive, mais également un `rollback` automatique puisqu'il est hors de question de laisser un programme s'exécuter seulement partiellement.

Mais la cohérence désigne également un état de la base considéré comme satisfaisant pour l'application, sans que cet état puisse être toujours spécifié par des contraintes SQL. Dans le cas par exemple de notre programme de réservation, la base est cohérente quand :

— le nombre de places prises pour un spectacle est le même que la somme des places réservées pour ce spectacle par les clients ;
— le solde de chaque client est supérieur à 0.

Il n'est pas facile d'exprimer cette contrainte avec les commandes DDL de SQL, mais on peut s'assurer qu'elle est respectée en écrivant soigneusement les procédures de mises à jour pour qu'elle tirent parti des propriétés ACID du système transactionnel.

Reprenons notre procédure de réservation, en supposant que la base est initialement dans un état cohérent (au sens donné ci-dessus de l'équilibre entre le nombre de places prises et le nombres de places réservées). Les propriétés d'atomicité (A) et de durabilité (D) garantissent que :

— la transaction s'effectue totalement, valide les deux opérations de mise à jour qui garantissent l'équilibre, et laisse donc après le `commit` la base dans un état cohérent ;
— la transaction est interrompue pour une raison quelconque, et la base revient alors à l'état initial, cohérent.

De plus toutes les contraintes définies dans le schéma sont respectées si la transaction arrive à terme (sinon le système l'annule).

Tout se passe bien parce que le programmeur a placé le `commit` au bon endroit. Imaginons maintenant qu'un `commit` soit introduit après le premier `update`. Si le second `update` soulève une erreur, le système effectue un `rollback` jusqu'au `commit` qui précède, et laisse donc la base dans un état incohérent –déséquilibre entre les places prises et les places réservées– du point de vue de l'application.

Dans ce cas l'erreur vient du programmeur qui a défini deux transactions au lieu d'une, et entraîné une validation à un moment où la base est dans un état intermédiaire. La leçon est simple : *tous les* `commit` *et* `rollback` *doivent être placés de manière à s'exécuter au moment où la base est dans un état cohérent*. Il faut toujours se souvenir qu'un `commit` ou un `rollback` marque la fin d'une transaction, et définit donc l'ensemble des opérations qui doivent s'exécuter solidairement (ou « atomiquement »).

Un défaut de cohérence peut enfin résulter d'un mauvais entrelacement des opérations concurrentes de deux transactions, dû à un niveau d'isolation insuffisant. Cet aspect sera illustré dans la prochaine section consacrée aux conséquences d'une absence de contrôle.

9.1.4 Quiz

9.2 S2 : Pratique des transactions

Supports complémentaires :

— Lien vers l'application de travaux pratiques en ligne
— Vidéo expliquant le fonctionnement de l'application « transactions »,
— Fichier de commandes pour tester les transactions sous un SGBD (MySQL, Oracle)

Pour cette session, nous vous proposons une mise en pratique consistant à constater directement les notions de base des transactions en interagissant avec un système relationnel. Vous avez deux possibilités : effectuer chez vous les commandes avec un système que vous avez installé localement (MySQL, Postgres, Oracle ou autre), ou utiliser l'application en ligne que nous mettons à disposition à l'adresse indiquée ci-dessus.

Quel que soit votre choix, n'hésitez pas à expérimenter et à chercher à comprendre le fonctionnement que vous constatez.

9.2.1 L'application en ligne « Transactions »

Ce TP est basé sur l'utilisation d'une base de données. Dans un premier temps, nous expliquons le contenu de la fenêtre d'interaction avec cette base.

Important : Vous pouvez à tout moment réinitialiser l'ensemble de la base en appuyant sur le bouton « Réinitialiser » en haut à droite de la fenêtre. Avant d'exécuter cette réinitialisation, vous devez sélectionner le niveau d'isolation parmi les 4 possibles : SERIALIZABLE, REPEATABLE READ, READ commitTED et READ UNcommitTED. Pour l'instant, conservez le niveau d'isolation par défaut, nous y reviendrons ultérieurement.

L'application est divisée en deux parties semblables, l'une à gauche, l'autre à droite, représentant deux sessions distinctes connectées à une même base. Ce dispositif permet de visualiser les interactions entre deux processus clients accédant de manière concurrente à des mêmes données. Nous détaillons maintenant les informations présentes dans chaque transaction.

Structure et contenu de la base

Le contenu de la base représente des clients achetant des places sur des vols. À peu de choses près, ce n'est qu'une variante de notre procédure de réservation. Son schéma est constitué de deux tables :

— Client(Id, Nom, Solde, Places)
— Vol (Id, Intitulé, Capacité, Réservations)

La table `Client` indique pour chaque client son nom, le solde de son compte et le nombre de places qu'il a réservées. Dans cette base de données, il y a deux clients : C1 et C2. La table `Vol` indique pour un vol donné la destination, la capacité et le nombre de réservations effectuées. Dans cette base, il n'y a qu'un seul vol V1. Il s'agit bien entendu d'un schéma très simplifié, destiné à étudier les mécanismes transactionnels.

Important : Sur cette base on définit la cohérence ainsi : **la somme des places réservées par les clients doit être égale à la somme des places réservées pour les vols.**

Vous noterez qu'à l'initialisation de la base, elle est dans un état cohérent. Chaque transaction, prise individuellement, préserve la cohérence de la base. Nous verrons que les mécanismes d'isolation ne garantissent pas que cela reste vrai en cas d'exécution concurrente.

L'affichage des tables montre, à tout moment, l'état de la base visible par une session. Nous affichons deux tables car chaque session peut avoir une vision différente, comme nous allons le voir.

Variables

En dessous des tables sont indiquées des variables utilisées par les transactions, soit comme paramètres, soit pour y stocker le résultat des requêtes. Par convention, le nom d'une variable est préfixé par « : ». Au départ, toutes les valeurs sont inconnues, sauf `:billet` qui représente le nombre de billets à réserver.

Actions

Quatre actions, qui correspondent à des requêtes utilisant les valeurs courantes des variables sont disponibles. Le code de chaque requête s'affiche lorsque vous passez la souris sur le bouton. Certaines requêtes peut être utilisées depuis la session S1 ou S2 selon qu'on les exécute à gauche ou à droite.

— Requête `select V1 into`

```
select capacité, réservations, tarif
into :capacité, :réservations, :tarif
from Vol where id=V1
```

Cette requête permet de stocker la capacité du vol et le nombre actuel de réservations dans les variables de la transaction.

— Requête `Select C1 into` :

```
select solde into :solde
from Client where id=C1
```

Cette requête permet de stocker le solde du client C1. Cette requête n'est accessible que depuis la transaction T1.

— Requête `select C2 into`

```
select solde into :solde
from Client where id=C2
```

Cette requête permet de stocker le solde du client C2. Cette requête n'est accessible que depuis la transaction T2.

— Requête `Update V1` :

```
update Vol set réservations=:réservations+billets
where id=V1 C
```

Cette requête permet de mettre à jour le nombre de réservations dans la table Vol, sur la base de la valeur des variables `:réservations` et `:billets` Cette requête est disponible depuis T1 et T2.

— Requête `Update C1` :

```
update Client set solde=:solde-:billets*:tarif
where id=C1
```

Cette requête permet de mettre à jour le solde du client C1 qui vient d'acheter les billets. Cette requête n'est disponible que depuis T1.

— Requête `Update C2`

```
update Client set solde=:solde-:billets*:tarif
where id=C2
```

Cette requête permet de mettre à jour le solde du client C2 qui vient d'acheter les billets. Cette requête n'est disponible que depuis T2.

— `Commit` : permet d'accepter toutes les mises à jour de la transaction.
— `Rollback` : permet d'annuler toutes les mises à jour depuis le dernier commit.

Une transaction se compose d'une suite d'actions, se terminant soit par un `commit`, soit par un `rollback`. Remarquez bien que si l'on exécute, dans l'ordre suggéré, les lectures puis les mises à jour, on obtient une transaction correcte qui préserve la cohérence de la base. Rien ne vous empêche d'effectuer les opérations dans un ordre différent si vous souhaitez effectuer un test en dehors de ceux donnés ci-dessous.

Historique

La liste des commandes effectuées, ainsi que les réponses du SGBD s'affichent au fur et à mesure de leur exécution. Celles-ci disparaissent lorsqu'on réinitialise l'application.

Nous vous invitons maintenant à regarder la vidéo et en reproduire vous-même le déroulement.

9.2.2 Quelques expériences avec l'interface en ligne

Voici quelques manipulations à faire avec l'interface en ligne. Vous pouvez suivre au préalable la vidéo indiquée en début de session, mais pratiquer vous-mêmes sera plus concret pour assimiler les principales leçons.

Leçon 1 : isolation des transactions

Effectuez une transaction de réservation à gauche (deux sélections, deux mises à jour), mais ne validez pas encore. Vous devriez constater que :

— les mises à jour sont *visibles* par la transaction qui les a effectuées,
— elles sont en revanche *invisibles* par toute autre transaction.

C'est la notion *d'isolation des transactions*. L'isolation est *complète* quand le résultat d'une transaction ne dépend pas de la présence ou non d'autres transactions. Nous verrons que l'isolation complète a des conséquences potentiellement pénalisantes et que les systèmes ne l'adoptent pas par défaut.

Leçon 2 : `commit` et `rollback`

Effectuez un `rollback` de la transaction en cours (à gauche). Vous devriez constater :

— que la base revient à son état initial ;
— que `commit` ou `rollback` n'ont plus aucun effet : nous sommes au début d'une nouvelle transaction ;
— pour la transaction de droite, rien n'a changé, tout se passe comme si celle de gauche n'avait pas existé.

Maintenant, lisez le vol à droite et constatez qu'il a zéro réservations. Recommencez la transaction à gauche, et effectuez un `commit`. Vous constatez que :

— il n'est plus possible d'annuler par `rollback`.
— les mises à jour sont *toujours* invisibles par l'autre transaction ;

Le fait de ne pas pouvoir annuler correspond à la notion de *durabilité* : le `commit` valide les données de manière définitive. Elles intègrent ce que nous appellerons *l'état de la base*, autrement dit l'ensemble des données validées, visibles par toutes les transactions.

Le fait que la transaction de droite ne puisse toujours pas voir les mises à jour est plus surprenant. C'est encore un effet de *l'isolation* : la transaction de droite a débuté avant celle de gauche, et elle continue durant toute son existence à voir la base telle qu'elle existait *au moment où elle a débuté*. C'est ce qu'indique le mode `repeatable read` : les mêmes lectures renvoient *toujours* le même résultat.

Commencez une autre transaction à droite : il suffit de faire `commit` ou `rollback`. Cette fois le nouvel état de la base apparaît.

Leçon 3 : les écritures concurrentes posent des verrous

Réinitialisez, toujours en mode `repeatable read`. Maintenant déroulez deux transactions en parallèle selon l'alternance suivante :

— on effectue les deux sélections à gauche, puis une mise à jour du vol ;
— on effectue les mêmes opérations à droite ;

On constate que la transaction de droit est bloquée sur la tentative d'écriture. *Il est impossible d'effectuer deux écritures concurrentes du même nuplet*. Il deviendrait en effet alors impossible de gérer correctement les possibilités de `commit` et de `rollback`. Un verrou est donc posé par le système sur un nuplet modifié par une transaction, et ce verrou empêche toute modification par une autre transaction.

Les verrous entraînent une mise en attente. Ils sont conservés jusqu'à la fin de la transaction qui les a posés (`commit` ou `rollback`). Effectuez un `rollback` à gauche, et constatez que la transaction de droite est libérée.

La leçon, c'est que les mises à jour peuvent potentiellement bloquer les autres transactions. Il faut de préférence les effectuer le plus tard possible dans une transaction pour limiter le temps de rétention.

Leçon 4 : isolation incomplète = incohérence possible

Réinitialisez, toujours en mode `repeatable read`. Maintenant déroulez deux transactions en parallèle selon l'alternance suivante :

— on effectue les deux sélections à gauche ;
— on effectue les deux sélections à droite ;
— on effectue les deux mises à jour à droite, et on valide ;
— on effectue les deux mises à jour à gauche, et on valide ;

On a effectué deux transactions indépendantes : l'une qui réserve 2 billets, l'autre 5. À l'arrivée, vous constaterez que les clients ont bien réservé 2+5=7 billets, mais que seulement 2 billets ont été réservés pour le vol.

La base est maintenant incohérente, alors que chaque transaction, individuellement, est correcte. *On constate un défaut de concurrence, dû à un niveau d'isolation incomplet*.

La leçon, c'est qu'un niveau d'isolation non maximal (c'est le cas ici) mène potentiellement (pas toujours, loin de là) à une incohérence. De plus cette incohérence est à peu près inexplicable, car elle ne survient que dans une situation très particulière.

Leçon 5 : isolation complète = blocages possibles

Si on veut assurer la cohérence, il faut donc choisir le mode d'isolation maximal, c'est-à-dire `serializable`. Réinitialisez en choisissant ce mode, et refaites la même exécution concurrente que précédemment.

Cette fois, on constate que le système se bloque et que l'une des transactions est rejetée. C'est la dernière leçon : en cas d'imbrication forte des opérations (ce qui encore une fois ne peut survenir que très rarement), on rencontre un risque *d'interblocage*.

En mode `serializable` le verrouillage est plus strict que dans le mode par défaut : *la lecture d'un nuplet par une transaction bloque les tentatives d'écriture par une autre transaction*. Les risques d'être mis en attente sont donc bien plus élevés. C'est la raison pour laquelle les systèmes ne choisissent pas ce mode par défaut.

Vous pouvez continuer à jouer avec la console, en essyant d'interpréter les résultats en fonction des remarques qui précèdent. Nous revenons en détail sur le fonctionnement d'un système transactionnel concurrent dans les prochaines sessions.

9.2.3 Mise en pratique directe avec un SGBD

Vous pouvez également pratiquer les transactions avec MySQL, Postgres ou Oracle. Un fichier de commandes vous est fourni ci-dessus. Voici deux sessions effectuées en concurrence sous ORACLE. Ces sessions s'effectuent avec l'utilitaire SQLPLUS qui permet d'entrer directement des requêtes sur la base comprenant les tables *Client* et *Spectacle* décrites en début de chapitre. Les opérations effectuées consistent à réserver, pour le même spectacle, 5 places pour la session 1, et 7 places pour la session 2.

Voici les premières requêtes effectuées par la session 1.

```
Session1>select * from Client;

ID_CLIENT NB_PLACES_RESERVEES      SOLDE
---------- -------------------- ----------
         1                    3       2000

Session1>select * from Spectacle;

ID_SPECTACLE NB_PLACES_OFFERTES NB_PLACES_LIBRES      TARif
------------ ------------------ ---------------- ----------
           1                250              200         10
```

On a donc un client et un spectacle. La session 1 augmente maintenant le nombre de places réservées.

```
Session1>update Client set nb_places_reservees = nb_places_reservees + 5
          where id_client=1;

1 row updated.

Session1>select * from Client;

ID_CLIENT NB_PLACES_RESERVEES      SOLDE
```

(suite sur la page suivante)

(suite de la page précédente)

```
---------- ------------------- ----------
         1                   8       2000

Session1>select * from Spectacle;

ID_SPECTACLE NB_PLACES_OFFERTES NB_PLACES_LIBRES      TARif
------------ ------------------ ---------------- ----------
           1                250              200         10
```

Après l'ordre `update`, si on regarde le contenu des tables *Client* et *Spectacle*, on voit bien l'effet des mises à jour. Notez que la base est ici dans un état instable puisqu'on n'a pas encore diminué le nombre de places libres. Voyons maintenant les requêtes de lecture pour la session 2.

```
Session2>select * from Client;

 ID_CLIENT NB_PLACES_RESERVEES      SOLDE
---------- ------------------- ----------
         1                   3       2000

Session2>select * from Spectacle;

ID_SPECTACLE NB_PLACES_OFFERTES NB_PLACES_LIBRES      TARif
------------ ------------------ ---------------- ----------
           1                250              200         10
```

Pour la session 2, la base est dans l'état initial. L'isolation implique que les mises à jour effectuées par la session 1 sont invisibles puisqu'elles ne sont pas encore validées. Maintenant la session 2 tente d'effectuer la mise à jour du client.

```
Session2>update Client set nb_places_reservees = nb_places_reservees + 7
               where id_client=1;
```

La transaction est mise en attente car, appliquant des techniques de verrouillage qui seront décrites plus loin, ORACLE a réservé le nuplet de la table *Client* pour la session 1. Seule la session 1 peut maintenant progresser. Voici la fin de la transaction pour cette session.

```
Session1>update Spectacle set nb_places_libres = nb_places_libres - 5
          where id_spectacle=1;

1 row updated.

Session1>commit;
```

Après le `commit`, la session 2 est libérée,

```
Session2>update Client set nb_places_reservees = nb_places_reservees + 7
              where id_client=1;
```

(suite sur la page suivante)

(suite de la page précédente)

```
1 row updated.
Session2>
Session2>select * from Client;

 ID_CLIENT NB_PLACES_RESERVEES      SOLDE
---------- ------------------- ----------
         1                  15       2000

Session2>select * from Spectacle;

ID_SPECTACLE NB_PLACES_OFFERTES NB_PLACES_LIBRES      TARif
------------ ------------------ ---------------- ----------
           1                250              195         10
```

Une sélection montre que les mises à jour de la session 1 sont maintenant visibles, puisque le `commit` les a validé définitivement. De plus on voit également la mise à jour de la session 2. Notez que pour l'instant la base est dans un état incohérent puisque 12 places sont réservées par le client, alors que le nombre de places libres a diminué de 5. La seconde session doit décider, soit d'effectuer la mise à jour de *Spectacle*, soit d'effectuer un `rollback`. En revanche il et absolument exclu de demander un `commit` à ce stade, même si on envisage de mettre à jour *Spectacle* ultérieurement. Si cette dernière mise à jour échouait, ORACLE ramènerait la base à l'état – incohérent – du dernier `commit`,

Voici ce qui se passe si on effectue le `rollback`.

```
Session2>rollback;

rollback complete.

Session2>select * from Client;

 ID_CLIENT NB_PLACES_RESERVEES      SOLDE
---------- ------------------- ----------
         1                   8       2000

Session2>select * from Spectacle;

ID_SPECTACLE NB_PLACES_OFFERTES NB_PLACES_LIBRES      TARif
------------ ------------------ ---------------- ----------
           1                250              195         10
```

Les mises à jour de la session 2 sont annulées : la base se retrouve dans l'état connu à l'issue de la session 1.

Ce court exemple montre les principales conséquences « visibles » du contrôle de concurrence effectué par un SGBD :

- chaque processus/session dispose d'une vision des données en phase avec les opérations qu'il vient d'effectuer ;
- les données modifiées mais non validées par un processus ne sont pas visibles pour les autres ;

— les accès concurrents à une même ressource peuvent amener le système à mettre en attente certains processus.

D'autre part le comportement du contrôleur de concurrence peut varier en fonction du niveau d'isolation choisi qui est, dans l'exemple ci-dessus, celui adopté par défaut dans ORACLE (`read committed`). Le choix (ou l'acceptation par défaut) d'un niveau d'isolation inapproprié peut entraîner diverses anomalies que nous allons maintenant étudier.

9.2.4 Quiz

9.3 S3 : effets indésirables des transactions concurrentes

Supports complémentaires :

— Diapositives: anomalies transactionnelles

— Vidéo sur les anomalies transactionnelles

Pour illustrer les problèmes potentiels en cas d'absence de contrôle de concurrence, ou de techniques insuffisantes, on va considérer un modèle d'exécution très simplifié, dans lequel il n'existe qu'une seule version de chaque nuplet (stocké par exemple dans un fichier séquentiel). Chaque instruction est effectuée par le système, dès qu'elle est reçue, de la manière suivante :

— quand il s'agit d'une instruction de lecture, on lit le nuplet dans le fichier et on le transmet au processus ;

— quand il s'agit d'une instruction de mise à jour, on écrit directement le nuplet dans le fichier en écrasant la version précédente.

Les problèmes consécutifs à ce mécanisme simpliste peuvent être classés en deux catégories : défauts d'isolation menant à des incohérences *–défauts de sérialisabilité–*, et difficultés dus à une mauvaise prise en compte des `commit` et `rollback`, ou *défauts de recouvrabilité*. Les exemples qui suivent ne couvrent pas l'exhaustivité des situations d'anomalies, mais illustrent les principaux types de problèmes.

9.3.1 Défauts de sérialisabilité

Considérons pour commencer que le système n'assure aucune isolation, aucun verrouillage, et ne connaît ni le `commit` ni le `rollback`. Même en supposant que toutes les exécutions concurrentes s'exécutent intégralement sans jamais rencontrer de panne, on peut trouver des situations où l'entrelacement des instructions conduit à des résultats différents de ceux obtenus par une exécution en série. De telles exécutions sont dites *non sérialisables* et aboutissent à des incohérences dans la base de données.

Les mises à jour perdues

Le problème de mise à jour perdue survient quand deux transactions lisent chacune une même donnée en vue de la modifier par la suite. Prenons à nouveau deux exécutions concurrentes du programme *Réservation*, désignées par T_1 et T_2. Chaque exécution consiste à réserver des places pour le même spectacle, mais pour deux clients distincts c_1 et c_2. L'ordre des opérations reçues par le serveur est le suivant :

$$r_1(s)r_1(c_1)r_2(s)r_2(c_2)w_2(s)w_2(c_2)w_1(s)w_1(c_1)$$

Donc on effectue d'abord les lectures pour T_1, puis les lectures pour T_2 enfin les écritures pour T_2 et T_1 dans cet ordre. Imaginons maintenant que l'on se trouve dans la situation suivante :

— il reste 50 places libres pour le spectacle *s*, c_1 et c_2 n'ont pour l'instant réservé aucune place ;
— T_1 veut réserver 5 places pour *s* ;
— T_2 veut réserver 2 places pour *s*.

Voici le résultat du déroulement imbriqué des deux exécutions *T_1(s, 5, c_1)* et *T_2(s, 2, c_2)*, en supposant que la séquence des opérations est celle donnée ci-dessus. On se concentre pour l'instant sur les évolutions du nombre de places vides.

— T_1 lit *s* et c_1 et constate qu'il reste 50 places libres ;
— T_2 lit *s* et c_2 et constate qu'il reste 50 places libres ;
— T_2 écrit *s* avec nb places = *50-2=48*.
— T_2 écrit le nouveau compte de c_2.
— T_1 écrit *s* avec nb places = *50-5=45*.
— T_1 écrit le nouveau compte de c_1.

À la fin de l'exécution, on constate un problème : il reste 45 places vides sur les 50 initiales alors que 7 places ont effectivement été réservées et payées. Le problème est clairement issu d'une mauvaise imbrication des opérations de T_1 et T_2 : T_2 lit et modifie une information que T_1 a déjà lue en vue de la modifier. La figure :numref :*trans_anom1* montre la superposition temporelle des deux transactions. On voit que T_1 et T_2 lisent chacun, dans leurs espaces mémoires respectifs, d'une copie de *S* indiquant 50 places disponibles, et que cette valeur sert au calcul du nombre de places restantes sans tenir compte de la mise à jour effectuée par l'autre transaction.

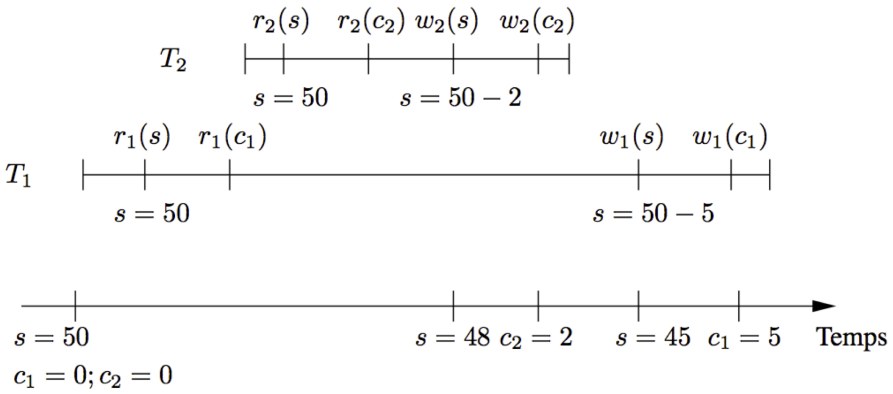

Fig. 9.4 – Exécution concurrente de T_1 et T_2

On arrive donc à une base de données incohérente alors que chaque transaction, prise isolément, est correcte, et qu'elles se sont toutes deux exécutées complètement.

Une solution radicale pour éviter le problème est d'exécuter *en série* T_1 et T_2. Il suffit pour cela de bloquer une des deux transactions tant que l'autre n'a pas fini de s'exécuter. On obtient alors l'exécution concurrente suivante :

$$r_1(s)r_1(c)w_1(s)w_1(c)r_2(s)r_2(c)w_2(s)w_2(c)$$

On est assuré dans ce cas qu'il n'y a pas de problème car T_2 lit la donnée écrite par T_1 qui a fini de s'exécuter et ne créera donc plus d'interférence. La Fig. 9.5 montre que la cohérence est obtenue ici par la lecture de s dans T_2 qui ramène la valeur 50 pour le nombre de places disponibles, ce qui revient bien à tenir compte des mises à jour de T_1.

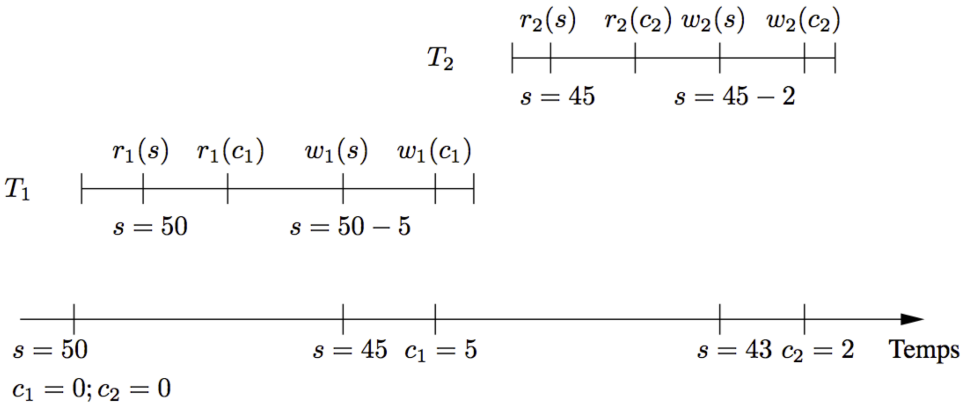

Fig. 9.5 – Exécution en série de T_1 et T_2

Cette solution de « concurrence zéro » est difficilement acceptable car elle revient à bloquer tous les processus sauf un. Dans un système où de très longues transactions (par exemple l'exécution d'un traitement lourd d'équilibrage de comptes) cohabitent avec de très courtes (des saisies interactives), les utilisateurs seraient extrêmement pénalisés.

Heureusement l'exécution en série est une contrainte trop forte, comme le montre l'exemple suivant.

$$r_1(s)r_1(c_1)w_1(s)r_2(s)r_2(c_2)w_2(s)w_1(c_1)w_2(c_2)$$

Suivons pas à pas l'exécution :

- T_1 lit *s* et c_1. Nombre de places libres : 50.
- T_1 écrit *s* avec nb places = *50-5=45*.
- T_2 lit *s*. Nombre de places libres : 45.
- T_2 lit c_2.
- T_2 écrit *s* avec nombre de places = *45-2=43*.
- T_1 écrit le nouveau compte du client c_1.
- T_2 écrit le nouveau compte du client c_2.

Cette exécution est correcte : on obtient un résultat strictement semblable à celui issu d'une exécution en série. Il existe donc des exécutions imbriquées qui sont aussi correctes qu'une exécution en série et qui permettent une meilleure concurrence. Le gain, sur notre exemple, peut paraître mineur, mais il il faut imaginer l'intérêt de débloquer rapidement de longues transactions qui ne rentrent en concurrence que sur une petite partie des nuplets qu'elles manipulent.

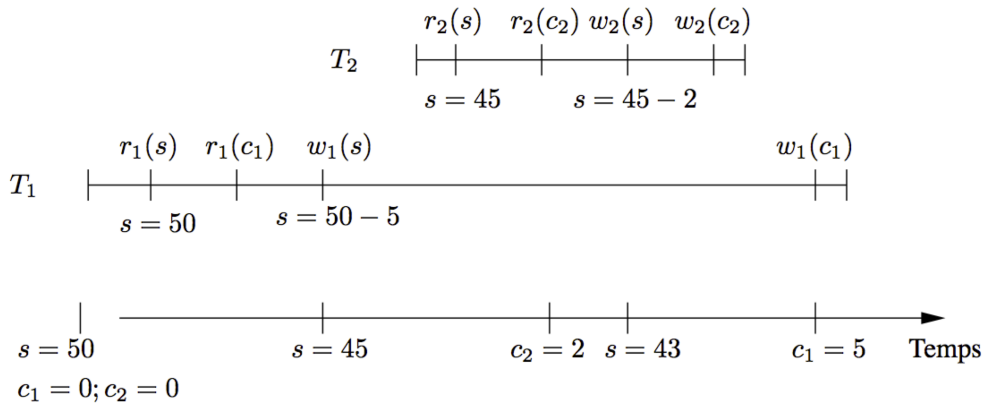

Fig. 9.6 – Exécution concurrente correcte de T_1 et T_2

On parle d'exécutions *sérialisables* pour désigner des exécutions concurrentes équivalentes à une exécution en série. Un des buts d'un système effectuant un contrôle de concurrence est d'obtenir de telles exécutions. Dans l'exemple qui précède, cela revient à mettre T_2 en attente tant que T_1 n'a pas écrit s. Nous verrons un peu plus loin par quelles techniques on peut automatiser ce genre de mécanisme.

Lectures non répétables

Voici un autre type de problème dû à l'interaction de plusieurs transactions : certaines modifications de la base peuvent devenir visibles *pendant* l'exécution d'une transaction *T* à cause des mises à jour effectuées *et* validées par d'autres transactions. Ces modifications peuvent rendre le résultat de l'exécution des requêtes en lecture effectuées par *T non répétables* : la première exécution d'une requête *q* renvoie un ensemble de nuplets différent d'une seconde exécution de *q* effectuée un peu plus tard, parce certain nuplets ont disparu ou sont apparus dans l'intervalle (on parle de *nuplets fantômes*).

Prenons le cas d'une procédure effectuant un contrôle de cohérence sur notre base de données : elle effectue tout d'abord la somme des places prises par les clients, puis elle compare cette somme au nombre de places réservées pour le spectacle. La base est cohérente si le nombre de places libres est égal au nombre de places réservées.

```
Lire tous les clients et effectuer la somme des places prises
Procédure Contrôle()
Début
  Lire le spectacle
  SI (Somme(places prises) <> places réservées)
    Afficher ("Incohérence dans la base")
 Fin
```

Une exécution de la procédure *Contrôle* se modélise simplement comme une séquence $r_c(c_1) \dots r_c(c_n) r_c(s)$ d'une lecture des *n* clients $\{c_1, \dots, c_n\}$ suivie d'une lecture de *s* (le spectacle). Supposons maintenant qu'on exécute cette procédure sous la forme d'une transaction T_1, en concurrence avec une réservation

$Res(c_1, s, 5)$, avec l'entrelacement suivant.

$$r_1(c_1)r_1(c_2)Res(c_2, s, 2)\ldots r_1(c_n)r_1(s)$$

Le point important est que l'exécution de $Res(c_2, s, 2)$ va augmenter de 2 le nombre de places réservées par c_2, et diminuer de 2 le nombre de places disponibles dans s. Mais la procédure T_1 a lu le spectacle *s après* la transaction *Res*, et le client c_2 *avant*. Seule une partie des mises à jour de *Res* sera donc visible, avec pour résultat la constatation d'une incohérence alors que ce n'est pas le cas.

La Fig. 9.7 résume le déroulement de l'exécution (en supposant deux clients seulement). Au début de la session 1, la base est dans un état cohérent. On lit 5 pour le client 1, 0 pour le client 2. À ce moment-là intervient la réservation T_2, qui met à jour le client 2 et le spectacle s. C'est cette valeur mise à jour que vient lire T_1.

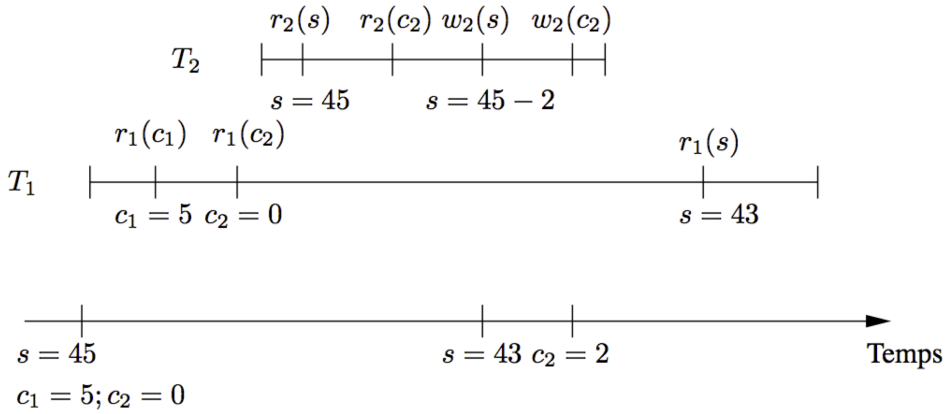

Fig. 9.7 – Exécution concurrente d'un contrôle et d'une réservation

Il faut noter que T_1 n'effectue pas de mise à jour et ne lit que des données validées. Il est clair que le problème vient du fait que la transaction T_1 accède, durant son exécution, à des versions différentes de la base et qu'un contrôle de cohérence ne peux être fiable dans ces conditions. On peut noter que cette exécution n'est pas sérialisable, puisque le résultat est clairement différent de celui obtenu par l'exécution successive de T_1 et de *Res*.

On désigne ce type de comportement par le terme de *lectures non répétables* (*non repeatable reads* en anglais). Si T_1 effectue deux lectures de s avant et après l'exécution de *Res*, elle constatera une modification.

De même toute insertion ou suppression d'un nuplet dans la table `Client` sera visible ou non selon que T_1 effectuera la lecture avant ou après la mise à jour concurrente. On parle alors de *nuplets fantômes*.

Les lectures non répétables et nuplets fantômes constituent un des effets désagréables d'une exécution concurrente. On pourrait considérer, à tort, que le risque d'avoir une telle interaction est faible. En fait l'exécution de requêtes sur une période assez longue est très fréquente dans les applications bases de données qui s'appuient sur des curseurs permettant de parcourir un résultat nuplet à nuplet, avec un temps de traitement de chaque nuplet qui peut alonger considérablement le temps seulement consacré au parcours du résultat.

9.3.2 Défauts de recouvrabilité

Le fait de garantir une imbrication sérialisable des exécutions concurrentes serait suffisant dans l'hypothèse où tous les programmes terminent normalement en validant les mises à jour effectuées. Malheureusement ce n'est pas le cas puisqu'il arrive que l'on doive annuler les opérations d'entrées sorties effectuées par un programme. Les anomalies d'une exécution concurrente dus au effets non contrôlés des `commit` et `rollback` constituent une seconde catégorie de problèmes qualifiés collectivement de *défauts de recouvrabilité*.

Nous allons maintenant étendre notre modèle d'exécution simplifié en introduisant les commandes `commit` et `rollback`. Attention : il ne s'agit que d'une version simplifiée de l'un des algorithmes possibles pour implanter les `commit` et `rollback` :

- le contrôleur conserve, chaque fois qu'une transaction T_i modifie un nuplet *t*, l'image de *t avant* la mise à jour, dénotée t_i^{ia} ;
- quand une transaction T_i effectue un `commit`, les images avant associées à T_i sont effacées ;
- quand une transaction T_i effectue un `rollback`, toutes les images avant sont écrites dans la base pour ramener cette dernière dans l'état du début de la transaction.

Imaginons par exemple que le programme de réservation soit interrompu après avoir exécuté les instructions suivantes :

$$r_1(s)r_1(c_1)w_1(s)$$

Au moment d'effectuer $w_1(s)$, notre système a conservé l'image avant modification s_1^{ia} du spectacle *s*. L'interruption intervient avant le `commit`, et la situation obtenue n'est évidemment pas satisfaisante puisqu'on on a diminué le nombre de places libres sans débiter le compte du client. Le `rollback` consiste ici à effectuer l'opération $w_1(s_1^{ia})$ pour réécrire l'image avant et revenir à l'état initial de la base.

L'implantation d'un tel mécanisme demande déjà un certain travail, mais cela ne suffit malheureusement toujours pas à garantir des exécutions concurrentes correctes, comme le montrent les exemples qui suivent.

Lectures sales

Revenons à nouveau à l'exécution concurrente de nos deux transactions T_1 et T_2, en considérant maintenant l'impact des validations ou annulations par `commit` ou `rollback`. Voici un premier exemple :

$$r_1(s)r_1(c_1)w_1(s)r_2(s)r_2(c_2)w_2(s)w_2(c_2)C_2w_1(c_1)R_1$$

Le nombre de places disponibles a donc été diminué par T_1 et repris par T_2, avant que T_1 n'annule ses réservations. On peut noter que cette exécution concurrente est sérialisable, au sens où l'ordre des opérations donne un résultat identique à une exécution en série.

Le problème vient ici du fait que T_1 est annulée *après* que la transaction T_2 a lu une information mise à jour par T_1, manipulé cette information, effectué une écriture, et enfin validé. On parle de « lectures sales » (*dirty read* en anglais) pour désigner l'accès par une transaction à des nuplets modifiés *mais non encore validés* par une autre transaction. L'exécution correcte du `rollback` est ici impossible, puisqu'on se trouve face au dilemne suivant :

- soit on écrit dans *s* l'image avant gérée par T_1, s_1^{ia}, mais on écrase du coup la mise à jour de T_2 alors que ce dernier a effectué un `commit` ;
- soit on conserve le nuplet *s* validé par T_2, et on annule seulement la mise à jour sur c_1, mais la base est alors incohérente.

On se trouve donc face à une exécution concurrente qui rend impossible le respect d'au moins une des deux propriétés transactionnelles requises : la durabilité (garantie du `commit`) ou l'atomicité (garantie du `rollback`). Une telle exécution est dite *non recouvrable*, et doit absolument être évitée.

La lecture sale transmet un nuplet modifié et non validé par une transaction (ici T_1) à une autre transaction (ici T_2). La première transaction, celle qui a effectué la lecture sale, devient donc dépendante du choix de la seconde, qui peut valider ou annuler. Le problème est agravé irrémédiablement quand la première transaction valide avant la seconde, comme dans l'exemple précédent, ce qui rend impossible une annulation globale.

Une exécution non-recouvrable introduit un conflit insoluble entre les `commit` effectués par une transaction et les `rollback` d'une autre. On pourrait penser à interdire à une transaction T_2 ayant effectué des lectures sales d'une transaction T_1 de valider avant T_1. On accepterait alors la situation suivante :

$$r_1(s)r_1(c_1)w_1(s)r_2(s)r_2(c_2)w_2(s)w_2(c_2)w_1(c_1)R_1$$

Ici, le `rollback` de T_1 intervient sans que T_2 n'ait validé. Il faut alors impérativement que le système effectue également un `rollback` de T_2 pour assurer la cohérence de la base : on parle *d'annulations en cascade* (noter qu'il peut y avoir plusieurs transactions à annuler).

Quoique acceptable du point de vue de la cohérence de la base, ce comportement est difficilement envisageable du point de vue de l'utilisateur qui voit ses transactions interrompues sans aucune explication liée à ses propres actions. Aucun SGBD ne pratique d'annulation en cascade. La seule solution est donc simplement d'interdire les *dirty read*. Nous verrons qu'il existe deux solutions : soit la lecture lit *l'image avant*, qui par définition est une valeur validée, soit on met en attente les lectures sur des nuplets en cours de modification.

Ecriture sale

Imaginons qu'une transaction *T* ait modifé un nuplet *t*, puis qu'un `rollback` intervienne. Dans ce cas, comme indiqué ci-dessus, il est nécessaire de restaurer la valeur qu'avait *t avant* le début de la transaction (« l'image avant »). Cela soulève des problèmes de concurrence illustrés par l'exemple suivant, toujours basé sur nos deux transactions T_1 et T_2. ci-dessous.

$$r_1(s)r_1(c_1)r_2(s)w_1(s)w_1(c_1)r_2(c_2)w_2(s)R_1w_2(c_2)C_2$$

Ici il n'y a pas de lecture sale, mais une « écriture sale » (*dirty write*) car T_2 écrit *s* après une mise à jour T_1 sur *s*, et sans que T_1 ait validé. Puis T_1 annule et T_2 valide. Que se passe-t-il au moment de l'annulation de T_1 ? On doit restaurer l'image avant connue de T_1, mais cela revient clairement à annuler la mise à jour de T_2.

On peut sans doute envisager des techniques plus sophistiquées de gestion des `rollback`, mais le principe de remplacement par l'image avant a le mérite d'être relativement simple à mettre en place, ce qui implique l'interdiction des écritures sales.

En résumé, on peut avoir des transactions sérialisables et non recouvrables et réciproquement. Le respect des propriétés ACID des transactions impose au SGBD d'assurer :

— la sérialisabilité des transactions ;
— la recouvrabilité dite *stricte*, autrement dit sans lectures ni écritures sales.

Les SGBD s'appuient sur un ensemble de techniques assez sophistiquées dont nous allons donner un aperçu ci-dessous. Il faut noter dès maintenant que le recours à ces techniques peut être pénalisant pour les performances (ou, plus exactement, la « fluiditéé des exécutions). Dans certains cas –fréquents– où le programmeur sait qu'il n'existe pas de risque lié à la concurrence, on peut relâcher le niveau de contrôle effectué par le système afin d'éviter des blocages inutiles.

9.3.3 Quiz

9.4 S4 : choisir un niveau d'isolation

Supports complémentaires :
— Diapositives: niveaux d'isolation
— Vidéo sur les niveaux d'isolation

Du point du programmeur d'application, l'objectif du contrôle de concurrence est de garantir la cohérence des données et d'assurer la recouvrabilité des transactions. Ces bonnes propriétés sont obtenues en choisissant un niveau d'isolation approprié qui garantit qu'aucune interaction avec un autre utilisateur ne viendra perturber le déroulement d'une transaction, empêcher son annulation ou sa validation.

Une option possible est de toujours choisir un niveau d'isolation maximal, garantissant la sérialisabilité des transactions, mais le mode `serializable` a l'inconvénient de ralentir le débit transactionnel pour des applications qui n'ont peut-être pas besoin de contrôles aussi stricts. On peut chercher à obtenir de meilleures performances en choisissant explicitement un niveau d'isolation moins élevé, soit parce que l'on sait qu'un programme ne posera jamais de problème de concurrence, soit parce les problèmes éventuels sont considérés comme peu importants par rapport au bénéfice d'une fluidité améliorée.

On considère dans ce qui suit deux exemples. Le premier consiste en deux exécutions concurrentes du programme *Réservation*, désignées respectivement par T_1 et T_2.

Exemple : concurrence entre mises à jour

Chaque exécution consiste à réserver des places pour le même spectacle, mais pour deux clients distincts c_1 et c_2. L'ordre des opérations reçues par le serveur est le suivant :

$$r_1(s)r_1(c_1)r_2(s)r_2(c_2)w_2(s)w_2(c_2)w_1(s)w_1(c_1)$$

Au départ nous sommes dans la situation suivante :
— il reste 50 places libres pour le spectacle s, c_1 et c_2 n'ont pour l'instant réservé aucune place ;
— T_1 veut réserver 5 places pour s ;
— T_2 veut réserver 2 places pour s.
Donc on effectue d'abord les lectures pour T_1, puis les lectures pour T_2 enfin les écritures pour T_2 et T_1 dans cet ordre. Aucun client n'a réservé de place.

Le second exemple prend le cas de la procédure effectuant un contrôle de cohérence sur notre base de données, uniquement par des lectures.

Exemple : concurrence entre lectures et mises à jour

La procédure *Contrôle* s'effectue en même temps que la procédure `Réservation` qui réserve 2 places pour le client c_2. L'ordre des opérations reçues par le serveur est le suivant (T_1 désigne le contrôle, T_2 la réservation) :

$$r_1(c_1)r_1(c_2)r_2(s)r_2(c_2)w_2(s)w_2(c_2)r_1(s)$$

Au départ le client c_1 a réservé 5 places. Il reste donc 45 places libres pour le spectacle. La base est dans un état cohérent.

9.4.1 Les modes d'isolation SQL

La norme SQL ANSI (SQL92) définit quatre modes d'isolation correspondant à quatre compromis différents entre le degré de concurrence et le niveau d'interblocage des transactions. Ces modes d'isolation sont définis par rapport aux trois types d'anomalies que nous avons rencontrés dans les exemples qui précèdent :

- *Lectures sales* : une transaction T_1 lit un nuplet mis à jour par une transaction T_2, *avant* que cette dernière ait validé ;
- *Lectures non répétables* : une transaction T_1 accède, en lecture ou en mise à jour, à un nuplet qu'elle avait déjà lu auparavant, alors que ce nuplet a été modifié entre temps par une autre transaction T_2 ;
- *Tuples fantômes* : une transaction T_1 lit un nuplet qui a été créé par une transaction T_2 *après* le début de T_1.

Tableau 9.1 – Niveaux d'isolation de la norme SQL

	Lectures sales	Lectures non répétables	Tuples fantômes
`read uncommitted`	Possible	Possible	Possible
`read committed`	Impossible	Possible	Possible
`repeatable read`	Impossible	Impossible	Possible
`serializable`	Impossible	Impossible	Impossible

Il existe un mode d'isolation par défaut qui varie d'un système à l'autre, le plus courant semblant être `read committed`.

Le premier mode (`read uncommitted`) correspond à l'absence de contrôle de concurrence. Ce mode peut convenir pour des applications non transactionnelles qui se contentent d'écrire « en vrac » dans des fichiers sans se soucier des interactions avec d'autres utilisateurs.

Avec le mode `read committed`, on ne peut lire que les nuplets validés, mais il peut arriver que deux lectures successives donnent des résultats différents. Le résultat d'une requête est cohérent par rapport à l'état de la base *au début de la requête*, Il peut arriver que deux lectures successives donnent des résultats différents si une autre transaction a modifié les données lues, et validé ses modifications. C'est le mode par défaut dans ORACLE par exemple.

Il faut bien être conscient que ce mode ne garantit pas l'exécution sérialisable. Le SGBD garantit par défaut l'exécution correcte des `commit` et `rollback` (recouvrabilité), mais pas la sérialisabilité. L'hypothèse effectuée implicitement est que le mode `serializable` est inutile dans la plupart des cas, ce qui est sans doute justifié, et que le programmeur saura le choisir explicitement quand c'est nécessaire, ce qui en revanche est loin d'être évident.

Le mode `repeatable read` (le défaut dans MySQL/InnoDB par exemple) garantit que le résultat d'une requête est cohérent par rapport à l'état de la base *au début de la transaction*. La réexécution de la même

requête donne toujours le même résultat. La sérialisabilité n'est pas assurée, et des nuplets peuvent apparaître s'ils ont étés insérés par d'autres transactions (les fameux « nuplets fantômes »).

Enfin le mode `serializable` assure les bonnes propriétés (sérialisabilité et recouvrabilité) des transactions et une isolation totale. Tout se passe alors comme si on travaillait sur une « image » de la base de données prise au début de la transaction. Bien entendu cela se fait au prix d'un risque assez élevé de blocage des autres transactions.

Le mode est choisi au début d'une transaction par la commande suivante.

```
set transaction isolation level <option>
```

Une autre option parfois disponible, même si elle ne fait pas partie de la norme SQL, est de spécifier qu'une transaction ne fera que des lectures. Dans ces conditions, on peut garantir qu'elle ne soulèvera aucun problème de concurrence et le SGBD peut s'épargner la peine de poser des verrous. La commande est :

```
set transaction read only
```

Il devient alors interdit d'effectuer des mises à jour jusqu'au prochain `commit` ou `rollback` : le système rejette ces instructions.

9.4.2 Le mode `read committed`

Le mode `read committed`, adopté par défaut dans ORACLE par exemple, amène un résultat incorrect pour nos deux exemples ! Ce mode ne pose pas de verrou en lecture, et assure simplement qu'une donnée lue n'est pas en cours de modification par une autre transaction. Voici ce qui se passe pour l'exemple *ex-conc-rw*.

— On commence par la procédure de contrôle qui lit le premier client, *r_1[c]*. Ce client a réservé 5 places. La procédure de contrôle lit c_2 qui n'a réservé aucune place. Donc le nombre total de places réservées est de 5.

— Puis c'est la réservation qui s'exécute, elle lit le spectacle, le client 2 (aucun de ces deux nuplets n'est en cours de modification). Le client c_2 réserve 2 places, donc au moment où la réservation effectue un `commit`, il y a 43 places libres pour le spectacle, 2 places réservées pour c_2.

— La session 1 (le contrôle) reprend son exécution et lit *s*. Comme *s* est validée on lit la valeur mise à jour juste auparavant par *Res*, et on trouve donc 43 places libres. La procédure de contrôle constate donc, à tort, une incohérence.

Le mode `read committed` est particulièrement inadapté aux longues transactions pour lesquelles le risque est fort de lire des données modifiées et validées après le début de la transaction. En contrepartie le niveau de verrouillage est faible, ce qui évite les bloquages.

9.4.3 Le mode `repeatable read`

Dans le mode `repeatable read`, chaque lecture effectuée par une transaction lit les données telles qu'elles étaient *au début de la transaction*. Cela donne un résultat correct pour l'exemple *ex-conc-rw*, comme le montre le déroulement suivant.

— On commence par la procédure de contrôle qui lit le premier client, *r_1[c]*. Ce client a réservé 5 places. La procédure de contrôle lit c_2 qui n'a réservé aucune place. Donc le nombre total de places réservées est de 5.

— Puis c'est la réservation qui s'exécute, elle lit le spectacle, le client 2 (aucun de ces deux nuplets n'est en cours de modification). Le client c_2 réserve 2 places, donc au moment où la réservation effectue une `commit`, il y a 43 places libres pour le spectacle, 2 places réservées pour c_2.

— La session 1 (le contrôle) reprend son exécution et lit s. Miracle ! La mise à jour de la réservation n'est pas visible car elle a été effectuée *après* le début de la procédure de contrôle. Cette dernière peut donc conclure justement que la base, *telle qu'elle était au début de la transaction*, est cohérente.

Ce niveau d'isolation est suffisant pour que les mises à jour effectuées par une transaction *T"* pendant l'exécution d'une transaction *T* ne soient pas visibles de cette dernière. Cette propriété est extrêmement utile pour les longues transactions, et elle a l'avantage d'être assurée sans aucun verrouillage.

En revanche le mode `repeatable read` ne suffit toujours pas pour résoudre le problème des mises à jour perdues. Reprenons une nouvelle fois l'exemple *ex-conc-trans*. Voici un exemple concret d'une session sous MySQL/InnoDB, SGBD dans lequel le mode d'isolation par défaut est `repeatable read`.

Note : Vous pouvez répéter ce déroulement avec notre interface en ligne.

C'est la première session qui débute, avec des lectures.

```
Session 1> START TRANSACTION;
Query OK, 0 rows affected (0,00 sec)

Session 1> select * from Spectacle where id_spectacle=1;
+--------------+--------------------+------------------+--------+
| id_spectacle | nb_places_offertes | nb_places_libres | tarif |
+--------------+--------------------+------------------+--------+
|            1 |                 50 |               50 | 10.00 |
+--------------+--------------------+------------------+--------+
1 row in set (0,01 sec)

Session 1> select * from Client where id_client=1;
+-----------+----------------------+--------+
| id_client | nb_places_reservees | solde |
+-----------+----------------------+--------+
|         1 |                   0 |   100 |
+-----------+----------------------+--------+
```

La session 1 constate donc qu'aucune place n'est réservée. Il reste 50 places libres. La session 2 exécute à son tour les lectures.

```
Session 2> START TRANSACTION;
Query OK, 0 rows affected (0,00 sec)

Session 2> select * from Spectacle where id_spectacle=1;
+--------------+--------------------+------------------+--------+
```

(suite sur la page suivante)

(suite de la page précédente)

```
| id_spectacle | nb_places_offertes | nb_places_libres | tarif |
+--------------+--------------------+------------------+-------+
|            1 |                 50 |               50 | 10.00 |
+--------------+--------------------+------------------+-------+
1 row in set (0,00 sec)

Session 2> select * from Client where id_client=2;
+-----------+----------------------+-------+
| id_client | nb_places_reservees | solde |
+-----------+----------------------+-------+
|         2 |                    0 |    60 |
+-----------+----------------------+-------+
```

Maintenant la session 2 effectue sa réservation de 2 places. Pensant qu'il en reste 50 avant la mise à jour, elle place le nombre 48 dans la table *Spectacle*.

```
Session 2> update Spectacle set nb_places_libres=48
       where id_spectacle=1;
Query OK, 1 row affected (0,00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

Session 2> update Client set solde=40, nb_places_reservees=2
         where id_client=2;
Query OK, 1 row affected (0,00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

Session 2> commit;
Query OK, 0 rows affected (0,00 sec)
```

Pour l'instant InnoDB ne dit rien. La session 1 continue alors. Elle aussi pense qu'il reste 50 places libres. La réservation de 5 places aboutit aux requêtes suivantes.

```
Session 1> update Spectacle set nb_places_libres=45 where id_spectacle=1;
Query OK, 1 row affected (0,00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

Session 1> update Client set solde=50, nb_places_reservees=5 where id_client=1;
Query OK, 1 row affected (0,00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

Session 1> commit;
Query OK, 0 rows affected (0,01 sec)

Session 1> select * from Spectacle where id_spectacle=1;
+--------------+--------------------+------------------+-------+
| id_spectacle | nb_places_offertes | nb_places_libres | tarif |
```

(suite sur la page suivante)

(suite de la page précédente)

```
+-------------+-------------------+-----------------+-------+
|           1 |                50 |              45 | 10.00 |
+-------------+-------------------+-----------------+-------+
1 row in set (0,00 sec)

Session 1> select * from Client;
+-----------+---------------------+--------+
| id_client | nb_places_reservees | solde |
+-----------+---------------------+--------+
|         1 |                   5 |    50 |
|         2 |                   2 |    40 |
+-----------+---------------------+--------+
```

La base est incohérente ! les clients ont réservé (et payé) en tout 7 places, mais le nombre de places libres n'a diminué que de 5. L'utilisation de InnoDB *ne garantit pas* la correction des exécutions concurrentes, du moins avec le niveau d'isolation par défaut.

Ce point est très souvent ignoré, et source de problèmes récurrents chez les organisations qui croient s'appuyer sur un moteur transactionnel assurant une cohérence totale, et constatent de manière semble-t-il aléatoire l'apparition d'incohérences et de déséquilibres dans leurs bases.

Note : La remarque est valable pour de nombreux autres SGBD, incluant ORACLE, dont le niveau d'isolation par défaut n'est pas maximal.

On soupçonne le plus souvent les programmes, à tort puisque c'est l'exécution concurrente qui, parfois, est fautive, et pas le programme. Il est extrêmement difficile de comprendre, et donc de corriger, ce type d'erreur.

9.4.4 Le mode `serializable`

Si on analyse attentivement l'exécution concurrente de l'exemple *ex-conc-trans*, on constate que le problème vient du fait que les deux transactions lisent, chacune de leur côté, une information (le nombre de places libres pour le spectacles) qu'elles s'apprêtent toutes les deux à modifier. Une fois cette information transférée dans l'espace mémoire de chaque processus, il n'existe plus aucun moyen pour ces transactions de savoir que cette information a changé dans la base, et qu'elles s'appuient donc sur une valeur incorrecte.

La seule chose qui reste à faire pour obtenir une isolation maximale est de s'assurer que cette situation ne se produit pas. C'est ce que garantit le mode `serializable`, au prix d'un risque de blocage plus important. On obtient ce niveau avec la commande suivante :

```
set transaction isolation level serializable;
```

Reprenons une dernière fois l'exemple *ex-conc-trans*, en mode sérialisable, avec MySQL/InoDB.

Note : Vous pouvez aussi choisir le niveau sérialisable et reproduire le scénario avec l'application en ligne

---

La session 1 commence par ses lectures.

```
Session 1> set TRANSACTION ISOLATION LEVEL serializable;
Query OK, 0 rows affected (0,04 sec)

Session 1> START TRANSACTION;
Query OK, 0 rows affected (0,00 sec)

Session 1> select * from Spectacle where id_spectacle=1;
+--------------+--------------------+------------------+-------+
| id_spectacle | nb_places_offertes | nb_places_libres | tarif |
+--------------+--------------------+------------------+-------+
|            1 |                 50 |               50 | 10.00 |
+--------------+--------------------+------------------+-------+
1 row in set (0,00 sec)

Session 1> select * from Client where id_client=1;
+-----------+----------------------+--------+
| id_client | nb_places_reservees | solde |
+-----------+----------------------+--------+
|         1 |                    0 |   100 |
+-----------+----------------------+--------+
```

Voici le tour de la session 2. Elle effectue ses lectures, et cherche à effectuer la première mise à jour.

```
Session 2> set TRANSACTION ISOLATION LEVEL serializable;
Query OK, 0 rows affected (0,00 sec)

Session 2> START TRANSACTION;
Query OK, 0 rows affected (0,00 sec)

Session 2> select * from Spectacle where id_spectacle=1;
+--------------+--------------------+------------------+-------+
| id_spectacle | nb_places_offertes | nb_places_libres | tarif |
+--------------+--------------------+------------------+-------+
|            1 |                 50 |               48 | 10.00 |
+--------------+--------------------+------------------+-------+
1 row in set (0,00 sec)

Session 2> select * from Client where id_client=2;
+-----------+----------------------+--------+
| id_client | nb_places_reservees | solde |
+-----------+----------------------+--------+
|         2 |                    0 |    60 |
+-----------+----------------------+--------+
1 row in set (0,00 sec)

Session 2> update Spectacle set nb_places_libres=48 where id_spectacle=1;
```

La transaction 2 est mise en attente car, en mode sérialisable, MySQL/InnoDB pose un verrou en lecture sur les lignes sélectionnées. La transaction 1 a donc verrouillé, en mode partagé, le spectacle et le client. La transaction 2 a pu lire le spectacle, et placer à son tour un verrou partagé, mais elle ne peut pas le modifier car cela implique la pose d'un verrou exclusif.

Que se passe-t-il alors du côté de la transaction 1 ? Elle cherche à faire la mise à jour du spectacle. Voici la réaction de InnoDB.

```
Session 1> update Spectacle set nb_places_libres=45 where id_spectacle=1;
ERROR 1213 (40001): Deadlock found when trying to get lock;
       try restarting transaction
```

Un interblocage (*deadlock*) a été détecté. La transaction 2 était déjà bloquée par la transaction 1. En cherchant à modifier le spectacle, la transaction 1 se trouve bloquée à son tour par les verrous partagées posés par la transaction 2.

En cas d'interblocage, les deux transactions peuvent s'attendre indéfiniment l'une l'autre. Le SGBD prend donc la décision d'annuler par `rollback` l'une des deux (ici, la transaction 1), en l'incitant à recommencer. La transaction 2 est libérée (elle garde ses verrous) et peut poursuivre son exécution.

Le mode `serializable` garantit la correction des exécutions concurrentes, au prix d'un risque de blocage et de rejet de certaines transactions. Ce risque, et ses effets désagréables (il faut resoumettre la transaction rejetée) expliquent qu'il ne s'agit pas du mode d'isolation par défaut. Pour les applications transactionnelles, il vaut sans doute mieux voir certaines transactions rejettées que courir un risque d'anomalie.

9.4.5 Verrouillage explicite

Certains systèmes permettent de poser explicitement des verrous, ce qui permet pour le programmeur averti de choisir un niveau d'isolation relativement permissif, tout en augmentant le niveau de verrouillage quand c'est nécessaire. ORACLE, PostgreSQL et MySQL proposent notamment une clause `FOR update` qui peut se placer à la fin d'une requête SQL, et dont l'effet est de réserver chaque nuplet lu en vue d'une prochaine modification.

Verrouillage des tables

Reprenons notre programme de réservation, et réécrivons les deux premières requêtes de la manière suivante :

```
...
select * into v_spectacle
from Spectacle
where id_spectacle=v_id_spectacle
FOR update;
...
select * into v_client from Client
where id_client=v_id_client
FOR update;
..
```

On annonce donc explicitement, dans le code, que la lecture d'un nuplet (le client ou le spectacle) sera suivie par la mise à jour de ce même nuplet. Le système pose alors un *verrou exclusif* qui réserve l'accès au nuplet, en lecture ou en mise à jour, à la transaction qui a effectué la lecture avec `FOR update`. Les verrous posés sont libérés au moment du `commit` ou du `rollback`.

Voici le déroulement de l'exécution pour l'exécution de l'exemple *ex-conc-trans* :

— T_1 lit s, *après l'avoir verrouillé exclusivement* ;
— T_1 lit c_1, et verrouille exclusivement ;
— T_2 veut lire s, et se trouve mise en attente ;
— T_1 continue, écrit s, écrit c_1, valide et libère les verrous ;
— T_2 est libéré et s'excéute.

On obtient l'exécution en série suivante.

$$r_1(s)r_1(c_1)w_1(s)w_1(c_1)C_1r_2(s)r_2(c_2)w_2(s)w_2(c_2)C_2$$

La déclaration, avec `FOR update` de l'intention de modifier un nuplet revient à le réserver et donc à empêcher un entrelacement avec d'autres transactions menant soit à un rejet, soit à une annulation autoritaire du SGBD.

Les SGBDs fournissent également des commandes de verrouillage explicite. On peut réserver, en lecture ou en écriture, une table entière. Un verrouillage en lecture est *partagé* : plusieurs transactions peuvent détenir un verrou en lecture sur la même table. Un verrouillage en écriture est *exclusif* : il ne peut y avoir aucun autre verrou, partagé ou exclusif, sur la table.

Voici un exemple avec MySQL dont un des moteurs de stockage, MyISAM, ne gère pas la concurrence. Il faut dont appliquer explicitement un verrouillage si on veut obtenir des exécutions concurrentes sérialisables. En reprenant l'exemple *ex-conc-trans* avec verrouillage exclusif (`WRITE`), voici ce que cela donne. La session 1 verrouille (en écriture), lit le spectacle puis le client 1.

```
Session 1> LOCK TABLES Client WRITE, Spectacle WRITE;
Query OK, 0 rows affected (0,00 sec)

Session 1> select * from Spectacle where id_spectacle=1;
+--------------+--------------------+------------------+--------+
| id_spectacle | nb_places_offertes | nb_places_libres | tarif |
+--------------+--------------------+------------------+--------+
|            1 |                 50 |               50 | 10.00 |
+--------------+--------------------+------------------+--------+
1 row in set (0,00 sec)

Session 1> select * from Client where id_client=1;
+-----------+----------------------+--------+
| id_client | nb_places_reservees | solde |
+-----------+----------------------+--------+
|         1 |                   0 |   100 |
+-----------+----------------------+--------+
```

La session 2 tente de verrouiller et est mise en attente.

```
Session 2> LOCK TABLES Client WRITE, Spectacle WRITE;
```

La session 1 peut finir ses mises à jour, et libère les tables avec la commande `UNLOCK TABLES`.

```
Session 1> update Spectacle set nb_places_libres=45
        where id_spectacle=1;
Query OK, 1 row affected (0,00 sec)

Session 1> update Client set solde=50, nb_places_reservees=5
         where id_client=1;
Query OK, 1 row affected (0,00 sec)

Session 1> UNLOCK TABLES;
```

La session 2 peut alors prendre le verrou, effectuer ses lectures et mises à jour, et libérer le verrou. Les deux transactions se sont effectuées en série, et le résultat est donc correct.

La granularité du verrouillage explicite avec `LOCK` est la table entière, ce qui est généralement considéré comme mauvais car un verrouillage au niveau de lignes permet à plusieurs transactions d'accéder à différentes lignes de la table.

Le verrouillage des tables est une solution de « concurrence zéro » qui est rarement acceptable car elle revient à bloquer tous les processus sauf un. Dans un système où de très longues transactions (par exemple l'exécution d'un traitement lourd d'équilibrage de comptes) cohabitent avec de très courtes (des saisies interactives), les utilisateurs sont extrêmement pénalisés. Pour ne rien dire du cas où on oublie de relâcher les verrous…

De plus, dans l'exemple *ex-conc-trans*, il n'existe pas de conflit sur les clients puisque les deux transactions travaillent sur deux lignes différentes c_1 et c_2. quand seules quelques lignes sont mises à jour, un verrouillage total n'est pas justifié.

Le verrouillage de tables peut cependant être envisagé dans le cas de longues transactions qui vont parcourir toute la table et souhaitent en obtenir une image cohérente. C'est par exemple typiquement le cas pour une sauvegarde. De même, si une longue transaction effectuant des mises à jour est en concurrence avec de nombreuses petites transactions, le risque d'interblocage, temporaire ou définitif (voir plus loin) est important, et on peut envisager de précéder la longue transaction par un verrouillage en écriture.

Verrouillage d'une ligne avec `FOR update`

Une alternative au mode `serializable` est la pause explicite de verrous sur les lignes que l'on s'apprête à modifier. La clause `FOR update` place un verrou exclusif sur les nuplets sélectionnés par un ordre `select`. Ces nuplets sont donc réservés pour une future modification : aucune autre transaction ne peut placer de verrou en lecture ou en écriture. L'intérêt est de ne pas réserver les nuplets qui sont simplement lues et non modifiées ensuite. Notez qu'en mode `serializable` toutes les lignes lues sont réservées, car le SGBD, contrairement au programmeur de l'application, ne peut pas deviner ce qui va se passer ensuite.

Voici l'exécution de l'exemple *ex-conc-trans*, en veillant à verrouiler les lignes que l'on va modifier.

— C'est la transaction 1 qui commence. Elle lit le spectacle et le client c_1 en posant un verrou exclusif avec la clause `FOR update`.

— Ensuite c'est la seconde transaction qui transmet ses commandes au serveur. Elle aussi cherche à placer des verrous (c'est normal, il s'agit de l'exécution du même code). Bien entendu elle est mise en attente puisque la session 1 a déjà posé un verrou exclusif.

— La session 1 peut continuer de s'exécuter. Le `commit` libère les verrous, et la transaction 2 peut alors conclure.

Au final les deux transactions se sont exécutées en série. La base est dans un état cohérent. L'utilisation de `FOR update` est un compromis entre l'isolation assurée par le système, et la déclaration explicite, par le programmeur, des données lues en vue d'être modifiées. Elle assure le maximum de fluidité pour une isolation totale, et minimise le risque d'interblocages. Le principal problème est qu'elle demande une grande discipline pendant l'écriture d'une application puisqu'il faut se poser la question, à chaque requête, des lignes que l'on va ou non modifier.

En résumé, il est de la responsabilité du programmeur, sur un SGBD n'adoptant pas le mode `SERIALISABLE` par défaut, de prendre lui-même les mesures nécessaires pour les transactions qui risquent d'aboutir à des incohérences en cas de concurrence sur les mêmes données. Ces mesures peuvent consister soit à passer en mode `serializable` pour ces transactions, soit à poser explicitement des verrous, en début de transaction, sur les données qui vont être modifiées ensuite.

9.4.6 Quiz

9.5 Exercices

Exercice Ex-transaction-1 : je comprends (et j'explique) les mécanismes transactionnels

Répondez aux questions suivantes clairement et concisément

— Pascal veut modifier un nuplet. Il s'aperçoit qu'il est mis en attente parce que ce nuplet est déjà en cours de modification par Julie. Comment lui expliqueriez-vous la justification de cette mise en attente ? À quel moment prendra-t-elle fin ?

— Justin a écrit un programme en mode `serializable` qui crée les fiches de paie de tout le personnel. Il vous annonce qu'il a décidé de n'effectuer qu'une seule fois l'ordre `commit`, quand toutes les fiches sont calculées. Comment lui expliqueriez-vous les inconvénients de ce choix ?

— Alphonsine lit un nuplet dans la table des voyageurs et s'aperçoit qu'il manque l'adresse. Elle demande à Barnabé d'effectuer la correction, et il le fait immédiatement. Pourtant, Alphonsine ne voit toujours pas l'adresse, même en réaffichant sans cesse son écran. La tension monte entre les deux. Que feriez-vous pour les aider ?

— Timothée veut surveiller les nouvelles réservations de billets d'avion. Il écrit une transaction qui effectue une boucle de lecture sur la table des vols et affiche le total des billets réservés. Or, le résultat affiché est toujours le même, alors que des billets sont bel et bien réservés. Que conseilleriez-vous à Timothée ?

Exercice Ex-transaction-2 : la sérialisabilité

Supposons une table *T(id, valeur)*, et la procédure suivante qui copie la valeur d'une ligne vers la valeur d'une autre :

```
/* Une procédure de copie */
create or replace procedure Copie (id1 INT, id2 INT) AS
```

(suite sur la page suivante)

(suite de la page précédente)

```
  -- Déclaration des variables
  val INT;
  begin
    -- On recherche la valeur de id1
    select valeur into val from T where id = id1

    -- On copie dans la ligne id2
    update T set valeur = val where id = id2

    -- Validation
    commit;
 end;
/
```

On prend deux transactions *Copie(A, B)* et *Copie(B,A)*, l'une copiant du nuplet A vers le nuplet B et l'autre effectuant la copie inverse. Initialement, la valeur de A est a et la valeur de B est b.

— Qu'est-ce qui caractérise une exécution concurrente correcte de ces deux transactions ?
Aide : une exécution est correcte si le résultat peut être obtenu par une exécution en série (exécution sérialisable). Etudiez les exécutions en série et déduisez-en la propriété générale du résultat.

— Voici une exécution concurrente de *Copie(A, B)* et *Copie(B,A)*

$$r_1(A)r_2(B)w_1(B)w_2(A)$$

En supposant qu'elle s'exécute sans contrôle de concurrence, avec une valeur initiale de A à a et une valeur de B à b, quel est l'état de la base à la fin ?

Conclusion : cette exécution est-elle sérialisable ?

— On se met en mode sérialisable. Dans ce mode, une transaction est mise en attente si elle tente de modifier un nuplet qui est en cours de lecture par une autre transaction.
Que se passe-t-il en mode sérialisable pour l'exécution concurrente précédente ?

Exercice Ex-transaction-3 : sérialisabilité et mode autocommit

On dispose d'une table $T (id, valeur)$. Initialement toutes les valeurs sont différentes. Voici une procédure qui échange les valeurs de 2 nuplets.

```
create or replace procedure Echange (id1 INT, id2 INT) AS
   -- Déclaration des variables
   val1, val2 integer;
   begin
       -- On recherche la valeur de id1 et de id2
       select valeur into val1 from T where id = id1
```

(suite sur la page suivante)

(suite de la page précédente)

```
      select valeur into val2 from T where id = id2

      -- On échange les valeurs
      update T set valeur = val1 where id = id2
      update T set valeur = val2 where id = id1

   end;
```

On est en mode `autocommit` : un `commit` a lieu après chaque requête SQL. Expliquez dans quel scénario l'exécution concurrente de deux procédures d'échange peut aboutir à ce que deux nuplets aient la même valeur.

Exercice Ex-transaction-4 : la sérialisabilité, suite

Supposons qu'un hôpital gère la liste de ses médecins dans une table (simplifiée) *Docteur(nom, garde)*, chaque médecin pouvant ou non être de garde. On doit s'assurer qu'il y a toujours au moins deux médecins de garde. La procédure suivante doit permettre de placer un médecin au repos en vérifiant cette contrainte.

```
/* Une procédure de gestion des gardes */

create or replace procedure HorsGarde (nomDocteur VARCHAR) AS

 -- Déclaration des variables
 val nb_gardes;

 begin
   -- On calcule le nombre de médecin de garde
   select count(*) into nb_gardes from Docteur where garde = true

   if (nb_gardes > 2) then
      update Docteur set garde = false where nom = nomDocteur;
      commit;
   endif
end;
/
```

En principe, cette procécure semble très correcte (et elle l'est). Supposons que nous ayons trois médecins, Philippe, Alice, et Michel, désignés par *p*, *a* et *m*, tous les trois de garde. Voici une exécution concurrente de deux transactions T_1 = *HorsGarde("Philippe")* et T_2 = *HorsGarde("Michel")*.

$$r_1(p)r_1(a)r_1(m)r_2(p)r_2(a)r_2(m)w_1(p)w_2(m)$$

Questions :

— Quel est, avec cette exécution, le nombre de médecins de garde constatés par T_1 et T_2

— Quel est le nombre de médecins de garde à la fin, quand T_1 et T_2 ont validé ?

— Au vu des réponses qui précèdent, expliquer pourquoi cette exécution concurrente n'est pas sérialisable (aide : définir la propriété de la base qui est respectée si les exécutions ont lieu en série, et constater qu'elle est violée ici).
— Mettons-nous en mode sérialisable, dans lequel l'écriture d'un nuplet est bloquée si une autre transaction a lu ce même nuplet (et ce, jusqu'à la fin de cette autre transaction). Que va-t-il se passer dans ce mode avec notre exécution concurrente ?

9.6 Atelier : réservons des places pour Philippe

Le but de cet atelier est de simuler des exécutions concurrentes avec un utilitaire de commandes SQL. Les instructions données ci-dessous ont été testées avec MySQL, elles fonctionnent probablement avec tout système relationnel, au prix éventuel de quelques ajustements.

9.6.1 Préparation

Il faut utiliser une application cliente qui permet de soumettre des commandes SQL au serveur. Pour MySQL, vous disposez de l'utilitaire `mysql`, ou du client graphique *MySQL workbench*, ou encore du client web phpMyAdmin. À vous de l'installer, de créer la base et le compte utilisateur.

Comme le support de cours montre comment réserver un même spectacle pour deux clients, nous allons simplement étudier une variante : réservation de deux spectacles pour le même client ! En d'autres termes, il s'agit d'une simple transposition de ce qui a été abondamment démontré ci-dessus, la mise en pratique en plus.

Voici tout d'abord les commandes de création des tables

```
create table Client (id_client integer not null,
                     nom varchar(30) not null,
                     nb_places_reservees integer not null,
                     solde integer not null,
                     primary key (id_client))
                      ;

create table Spectacle (id_spectacle integer not null,
                        titre varchar(30) not null,
                        nb_places_offertes integer not null,
                        nb_places_libres integer not null,
                        tarif decimal(10,2) not null,
                        primary key (id_spectacle))
                     ;
```

Insérez des données dans la base (et refaites cette insertion quand vous souhaitez la réinitialiser). Vous pouvez exécuter en bloc les commandes suivantes :

```
set autocommit = 0;
delete from Client;
delete from Spectacle;
insert intoClient values (1, 'Philippe', 0, 2000);
insert intoClient values (2, 'Julie', 0, 350);
insert intoSpectacle values (1, 'Ben hur', 250, 50, 50);
insert intoSpectacle values (2, 'Tartuffe', 120, 30, 30);
commit;
```

Vous pouvez alors ouvrir deux connexions simultanées à votre base. La première chose à faire est de s'assurer que l'on n'est pas en mode `autocommit` (mode où chaque mise à jour se termine par un `commit`, autrement dit un mode anti-transactionnel). Pour MySQL la commande est :

```
set autocommit = 0;
```

Nous appellerons les deux connexions Session1 et Session2. Le second réglage est le niveau d'isolation. Choisissez une des commandes ci-dessous.

```
set session transaction isolation level read uncommitted;
set session transaction isolation level read committed;
set session transaction isolation level repeatable read;
set session transaction isolation level serializable;
```

Et pour finir, il faudra, pour simuler une réservation, exécuter pas à pas les commandes de sélection et de mise à jour. Voici quelques exemple à reprendre et adapter.

```
select * from Client where id_client=1;
select * from Spectacle where id_spectacle=2;

update Client set nb_places_reservees = 0 + 2 where id_client=2;
update Spectacle set nb_places_libres = 50 - 2 where id_spectacle=1;
```

9.6.2 Déroulement

Le but est de réserver des places pour Philippe : 2 places pour Ben Hur, et 4 pour Tartuffe. On a donc deux exécutions de la procédure de réservation, l'une à dérouler dans la session 1, l'autre dans la session 2.

- Commencez par écrire les requêtes effectuées par chaque transaction
- Effectuez les deux transactions l'une après l'autre (en série). Quel est l'état de la base à la fin ? Cela vous satisfait-il ?
- Maintenant, en mode `read committed` ou `repeatable read`, déroulez des deux transactions dans l'ordre suivant : la transaction 1 fait ses lectures ; la transaction 2 fait ses lectures ; la transaction 1 fait ses écritures, la transaction 2 fait ses écritures.
  Quel est l'état de la base à la fin ? Conclusion ?
- Recommencez en mode `serialisable`.

Vous devriez arriver à des conclusions déjà largement détaillées dans le cours. À vous de faire le lien ! Voici une étude de cas qui va nous permettre de récapituler à peu près tout le contenu de ce cours. Nous étudions la mise en œuvre d'une base de données destinée à soutenir une application de messagerie (extrêmement

simplifiée bien entendu). Même réduite aux fonctionnalités de base, cette étude mobilise une bonne partie des connaissances que vous devriez avoir assimilées. Vous pouvez vous contenter de lire le chapitre pour vérifier votre compréhension. Il est sans doute également profitable d'essayer d'appliquer les commandes et scripts présentés.

CHAPITRE 10

Une étude de cas

Imaginons donc que l'on nous demande de concevoir et d'implanter un système de messagerie, à intégrer par exemple dans une application web ou mobile, afin de permettre aux utilisateurs de communiquer entre eux. Nous allons suivre la démarche complète consistant à analyser le besoin, à en déduire un schéma de données adapté, à alimenter et interroger la base, et enfin à réaliser quelques programmes en nous posant, au passage, quelques questions relatives aux aspects transactionnels ou aux problèmes d'ingénierie posés par la réalisation d'applications liées à une base de données.

10.1 S1 : Expression des besoins, conception

Supports complémentaires :

— Diapositives: conception
— Vidéo sur la conception

Dans un premier temps, il faut toujours essayer de clarifier les besoins. Dans la vie réelle, cela implique beaucoup de réunions, et d'allers-retours entre la rédaction de documents de spécification et la confrontation de ces spécifications aux réactions des futurs utilisateurs. La mise en place d'une base de données est une entreprise délicate car elle engage à long terme. Les tables d'une base sont comme les fondations d'une maison : il est difficile de les remettre en cause une fois que tout est en place, sans avoir à revoir du même coup tous les programmes et toutes les interfaces qui accèdent à la base.

Voici quelques exemples de besoins, exprimés de la manière la plus claire possible, et orientés vers les aspects-clé de la conception (notamment la détermination des entités, de leurs liens et des cardinalités de participation).

— « Je veux qu'un utilisateur puisse envoyer un message à un autre »
— « Je veux qu'il puisse envoyer à *plusieurs* autres »
— « Je veux savoir qui a envoyé, qui a reçu, quel message »

— « Je veux pouvoir répondre à un message en le citant »

Ce n'est que le début. On nous demandera sans doute de pouvoir envoyer des fichiers, de pouvoir choisir le mode d'envoi d'un message (destinataire principal, copie, copie cachée, etc.), de formatter le message ou pas, etc

On va s'en tenir là, et commencer à élaborer un schéma entité-association. En première approche, on obtient celui de la Fig. 10.1.

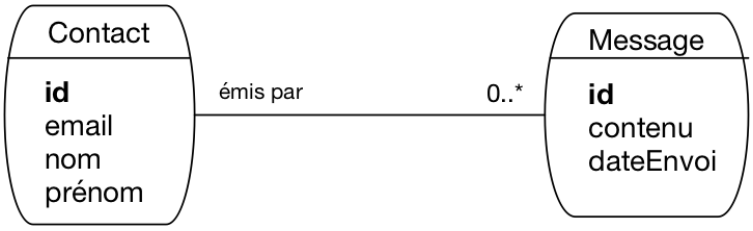

Fig. 10.1 – Le schéma de notre messagerie, première approche

Il faut *nommer* les entités, définir leur *identifiant* et les *cardinalités* des associations. Ici, nous avons une première ébauche qui semble raisonnable. Nous représentons des entités qui émettent des messages. On aurait pu nommer ces entités « Personne » mais cela aurait semblé exclure la possibilité de laisser une *application* envoyer des messages (c'est le genre de point à clarifier lors de la prochaine réunion). On a donc choisi d'utiliser le terme plus neutre de « Contact ».

Même si ces aspects terminologiques peuvent sembler mineurs, ils impactent la compréhension du schéma et peuvent donc mener à des malentendus. Il est donc important d'être le plus précis possible.

Le schéma montre qu'un contact peut envoyer plusieurs messages, mais qu'un message n'est envoyé que par un seul contact. Il manque sans doute les destinataires du message. On les ajoute donc dans le schéma de la Fig. 10.2.

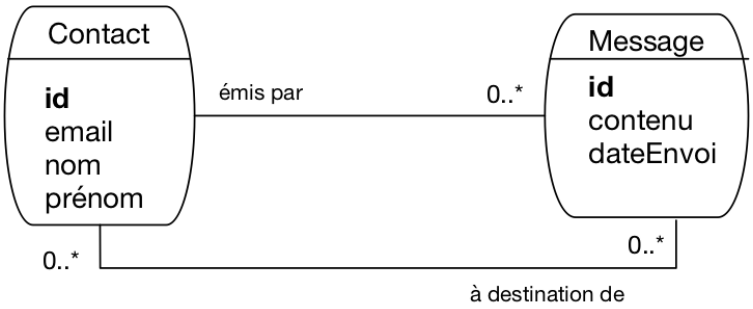

Fig. 10.2 – Le schéma de notre messagerie, avec les destinataires

Ici, on a considéré qu'un message peut être envoyé à plusieurs contacts (cela fait effectivement partie des besoins exprimés, voir ci-dessus). Un contact peut évidemment recevoir plusieurs messages. Nous avons donc une première association plusieurs-plusieurs. On pourrait la réifier en une entité nommée, par exemple « Envoi ». On pourrait aussi qualifier l'association avec des attributs propres : le mode d'envoi par exemple serait à placer comme caractéristique de l'association, et pas du message car un même message peut être envoyé dans des modes différents en fonction du destinataire. Une des attributs possible de l'association est d'ailleurs la date d'envoi : actuellement elle qualifie le message, ce qui implique qu'un message est envoyé *à la même date* à tous les destinataires. C'est peut-être (sans doute) trop restrictif.

On voit que, même sur un cas aussi simple, la conception impose de se poser beaucoup de questions. Il faut y répondre en connaissance de cause : la conception, c'est un ensemble de choix qui doivent être explicites et informés.

Il nous reste à prendre en compte le fait que l'on puisse répondre à un message. On a choisi de représenter de manière générale le fait qu'un message peut être le successeur d'un autre, ce qui a l'avantage de permettre la gestion du cas des renvois et des transferts. On obtient le schéma de la Fig. 10.3, avec une association réflexive sur les messages.

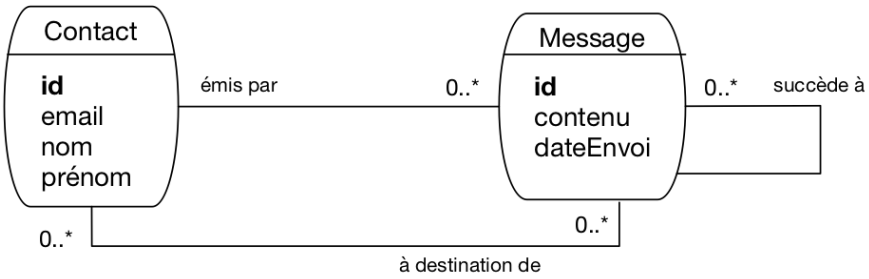

Fig. 10.3 – Le schéma complet de notre messagerie

Un schéma peut donc avoir plusieurs successeurs (on peut y répondre plusieurs fois) mais un seul prédécesseur (on ne répond qu'à un seul message). On va s'en tenir là pour notre étude.

À ce stade il n'est pas inutile d'essayer de construire un exemple des données que nous allons pouvoir représenter avec cette modélisation (une « instance » du modèle). C'est ce que montre par exemple la Fig. 10.4.

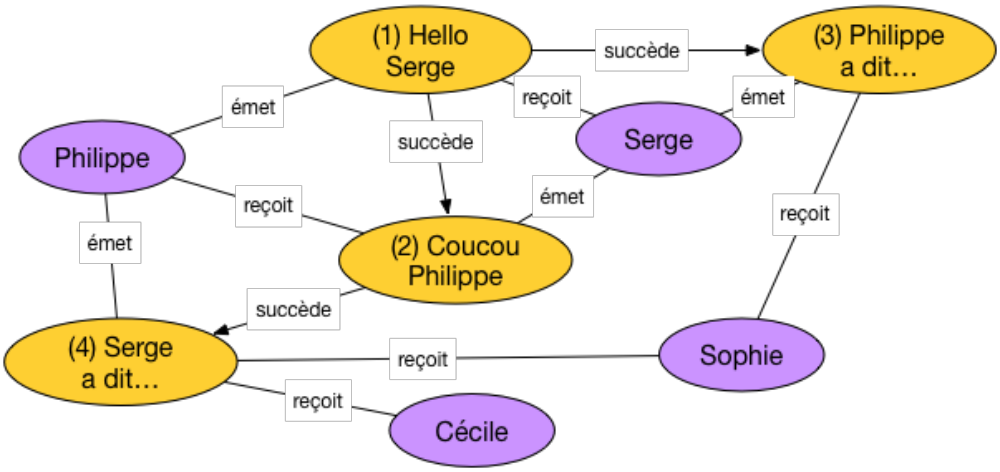

Fig. 10.4 – Une instance (petite mais représentative) de notre messagerie

Sur cet exemple nous avons quatre contacts et quatre messages. Tous les cas envisagés sont représentés :

- un contact peut émettre plusieurs messages (c'est le cas pour Serge ou Philippe)
- un contact peut aussi recevoir plusieurs messages (cas de Sophie)
- un message peut être envoyé à plusieurs destinataires (cas du message 4, « Serge a dit… », transmis à Sophie et Cécile)
- un message peut être le successeur d'un (unique) autre (messages 2, 3, 4) ou non (message 1)
- un message peut avoir plusieurs successeurs (message 1) mais toujours un seul prédécesseur.

Prenez le temps de bien comprendre comment les propriétés du modèle sont représentées sur l'instance.

Nous en restons là pour notre étude. Cela n'exclut en aucun cas d'étendre le modèle par la suite (c'est inévitable, car des besoins complémentaires arrivent toujours). Il est facile
- d'ajouter des attributs aux entités ou aux associations existantes ;
- d'ajouter de nouvelles entités ou associations.

En revanche, il est difficile de revenir sur les choix relatifs aux entités ou aux associations déjà définies. C'est une très bonne raison pour faire appel à toutes les personnes concernées, et leur faire valider les choix effectués (qui doivent être présentés de manière franche et complète).

10.1.1 Quiz

10.2 S2 : schéma de la base

Supports complémentaires :
- Diapositives: schéma de la base
- Vidéo sur le schéma de la base

Maintenant, nous sommes prêts à implanter la base en supposant que le schéma E/A de la Fig. 10.4 a été validé. Avec un peu d'expérience, la production des commandes de création des tables est directe. Prenons une dernière fois le temps d'expliquer le sens des règles de passage.

Note : Pour appliquer les commandes qui suivent, vous devez disposer d'un accès à un serveur. Une base doit être créée. Par exemple :

```
create database Messagerie
```

Et vous disposez d'un utilisateur habilité à créer des tables dans cette base. Par exemple :

```
grant all on Messagerie.* to athénaïs identified by 'motdepasse'
```

On raisonne en terme de dépendance fonctionnelle. Nous avons tout d'abord celles définies par les entités.
- $idContact \rightarrow nom, prnom, email$
- $idMessage \rightarrow contenu, dateEnvoi$

C'est l'occasion de vérifier une dernière fois que tous les attributs mentionnés sont atomiques (`email` par exemple représente *une seule* adresse électronique, et pas une liste) et qu'il n'existe pas de dépendance fonctionnelle non explicitée. Ici, on peut trouver la DF suivante :
- $email \rightarrow idContact, nom, prnom$

Elle nous dit que `email` est une clé candidate. Il faudra le prendre en compte au moment de la création du schéma relationnel.

Voici maintenant les dépendances données par les associations. La première lie un message au contact qui l'a émis. On a donc une dépendance entre les identifiants des entités.
- $idMessage \rightarrow idContact$

Un fois acquis que la partie droite est l'identifiant du contact, le nommage est libre. Il est souvent utile d'introduire dans ce nommage la signification de l'association représentée. Comme il s'agit ici de *l'émission* d'un message par un contact, on peut représenter cette DF avec un nommage plus explicite.

— $idMessage \rightarrow idEmetteur$

La seconde DF correspond à l'association plusieurs-à-un liant un message à celui auquel il répond. C'est une association réflexive, et pour le coup la DF $idMessage \rightarrow idMessage$ n'aurait pas grand sens. On passe donc directement à un nommage représentatif de l'association.

— $idMessage \rightarrow idPrdcesseur$

Etant entendu que `idPrédécesseur` est l'identifiant d'un message. Nous avons les DF, il reste à identifier les clés. Les attributs `idContact` et `idMessage` sont les clés primaires, `email` est une clé secondaire, et nous ne devons pas oublier la clé définie par l'association plusieurs-plusieurs représentant l'envoi d'un message. Cette clé est la paire `(idContact, idMessage)`, que nous nommerons plus explicitement `(idDestinataire, idMessage)`.

Voilà, nous appliquons l'algorithme de normalisation qui nous donne les relations suivantes :

— Contact (**idContact**, nom, prénom, email)
— Message (**idMessage**, contenu, dateEnvoi, *idEmetteur*, *idPrédécesseur*)
— Envoi (**idDestinataire**, **idMessage**)

Les clés primaires sont en gras, les clés étrangères (correspondant aux attributs issus des associations plusieurs-à-un) en italiques.

Nous sommes prêts à créer les tables. Voici la commande de création de la table `Contact`.

```
create table Contact (idContact integer not null,
                  nom varchar(30) not null,
                  prénom varchar(30)  not null,
                  email varchar(30) not null,
                  primary key (idContact),
                  unique (email)
               );
```

On note que la clé secondaire `email` est indiquée avec la commande `unique`. Rappelons pourquoi il semble préférable de ne pas la choisir pour clé primaire : la clé primaire d'une table est référencée par des clés étrangères dans d'autres tables. Modifier la clé primaire implique de modifier de manière synchrone les clés étrangères, ce qui peut être assez délicat.

Voici la table des messages, avec ses clés étrangères.

```
create table Message (
    idMessage  integer not null,
    contenu text not null,
    dateEnvoi   datetime,
    idEmetteur int not null,
    idPrédecesseur int,
    primary key (idMessage),
    foreign key (idEmetteur)
          references Contact(idContact),
    foreign key (idPrédecesseur)
```

(suite sur la page suivante)

(suite de la page précédente)

```
        references Message(idMessage)
)
```

L'attribut `idEmetteur`, clé étrangère, est déclaré `not null`, ce qui impose de *toujours* connaître l'émetteur d'un message. Cette contrainte, dite « de participation » semble ici raisonnable.

En revanche, un message peut ne pas avoir de prédécesseur, et `idPrédécesseur` peut donc être à `null`, auquel cas la contrainte d'intégrité référentielle ne s'applique pas.

Et pour finir, voici la table des envois.

```
create table Envoi (
    idDestinataire  integer not null,
    idMessage  integer not null,
    primary key (idDestinataire, idMessage),
    foreign key (idDestinataire)
           references Contact(idContact),
    foreign key (idMessage)
           references Message(idMessage)
)
```

C'est la structure typique d'une table issue d'une association plusieurs-plusieurs. La clé est composite, et chacun de ses composants est une clé étrangère. On remarque que la structure de la clé empêche d'un même message soit envoyé deux fois à un même destinataire (plus précisément : on ne saurait pas représenter des envois multiples). C'est un choix dont l'origine remonte à la conception E/A.

10.2.1 Quiz

10.3 S3 : requêtes

Supports complémentaires :

— Diapositives: requêtes

— Vidéo sur les requêtes

Pour commencer, nous devons peupler la base. Essayons de créer l'instance illustrée par la Fig. 10.4. Les commandes qui suivent correspondent aux deux premiers messages, les autres sont laissés à titre d'exercice.

Il nous faut d'abord au moins deux contacts.

```
insert into Contact (idContact, prénom, nom,  email)
  values (1, 'Serge', 'A.', 'serge.a@inria.fr');
insert into Contact (idContact, prénom, nom,  email)
  values (4, 'Philippe', 'R.', 'philippe.r@cnam.fr');
```

L'insertion du premier message suppose connue l'identifiant de l'emetteur. Ici, c'est Philippe R., dont l'identifiant est 4. Les messages eux-mêmes sont (comme les contacts) identifiés par un numéro séquentiel.

```
insert into Message (idMessage, contenu, idEmetteur)
values (1, 'Hello Serge', 4);
```

Attention, la contrainte d'intégrité référentielle sur la clé étrangère implique que l'émetteur (Philippe) doit exister au moment de l'insertion du message. Les insertions ci-dessus dans un ordre différent entraineraient une erreur.

Note : Laisser l'utilisateur fournir lui-même l'identifiant n'est pas du tout pratique. Il faudrait mettre en place un mécanisme de séquence, dont le détail dépend (malheureusement) du SGBD.

Et la définition du destinataire.

```
insert into Envoi (idMessage, idDestinataire) values (1, 1);
```

La date d'envoi n'est pas encore spécifiée (et donc laissée à `null`) puisque la création du message dans la base ne signifie pas qu'il a été envoyé. Ce sera l'objet des prochaines sessions.

Nous pouvons maintenant insérer le second message, qui est une réponse au premier et doit donc référencer ce dernier comme prédécesseur. Cela suppose, encore une fois, de connaître son identifiant.

```
insert into Message (idMessage, contenu, idEmetteur, idPrédecesseur)
values (2, 'Coucou Philippe', 1, 1);
```

On voit que la plupart des données fournies sont des identifiants divers, ce qui rend les insertions par expression directe de requêtes SQL assez pénibles et surtout sujettes à erreur. Dans le cadre d'une véritable application, ces insertions se font après saisie via une interface graphique qui réduit considérablement ces difficultés.

Nous n'avons plus qu'à désigner le destinataire de ce deuxième message.

```
insert into Envoi (idMessage, idDestinataire)
values (2, 4);
```

Bien malin qui, en regardant ce nuplet, pourrait deviner de quoi et de qui on parle. Il s'agit purement de la définition d'un lien entre un message et un contact.

Voici maintenant quelques exemples de requêtes sur notre base. Commençons par chercher les messages et leur émetteur.

```
select idMessage, contenu, prénom, nom
from Message as m,  Contact as c
where m.idEmetteur = c.idContact
```

Comme souvent, la jointure associe la clé primaire (de `Contact`) et la clé étrangère (dans le message). La jointure est l'opération inverse de la normalisation : elle regroupe, là où la normalisation décompose.

On obtient le résultat suivant (en supposant que la base correspond à l'instance de la Fig. 10.4).

idMessage	contenu	prénom	nom
1	Hello Serge	Philippe	R
2	Coucou Philippe	Serge	A
3	Philippe a dit …	Serge	A
4	Serge a dit …	Philippe	R

Cherchons maintenant les messages et leur prédécesseur.

```
select m1.contenu as 'Contenu', m2.contenu as 'Prédecesseur'
from Message as m1,  Message as m2
where m1.idPrédecesseur = m2.idMessage
```

Ce qui donne :

Contenu	Prédecesseur
Coucou Philippe	Hello Serge
Philippe a dit …	Hello Serge
Serge a dit …	Coucou Philippe

Quelle est la requête (si elle existe…) qui donnerait la liste complète des prédécesseurs d'un message ? Réflechissez-y, la question est épineuse et fera l'objet d'un travail complémentaire.

Et voici une requête d'agrégation : on veut tous les messages envoyés à plus d'un contact.

```
select m.idMessage, contenu, count(*) as 'nbEnvois'
from Message as m, Envoi as e
where m.idMessage = e.idMessage
group by idMessage, contenu
having nbEnvois > 1
```

Si une requête est un tant soit peu compliquée et est amenée à être exécutée souvent, ou encore si le résultat de cette requête est amené à servir de base à des requêtes complémentaires, on peut envisager de créer une vue.

```
create view EnvoisMultiples as
select m.idMessage, contenu, count(*) as 'nbEnvois'
from Message as m, Envoi as e
where m.idMessage = e.idMessage
group by idMessage, contenu
having nbEnvois > 1
```

Pour finir, un exemple de mise à jour : on veut supprimer les messages anciens, disons ceux antérieurs à 2015.

```
delete from Message; where year(dateEnvoi) < 2015
```

Malheureusement, le système nous informe qu'il a supprimé tous les messages :

```
All messages deleted. Table message is now empty..
```

Que s'est-il passé ? Un point virgule mal placé (vérifiez). Est-ce que tout est perdu ? Non, réfléchissez et trouvez le bon réflexe. Cela dit, les mises à jour et destructions devraient être toujours effectuées dans un cadre très contrôlé, et donc par l'intermédiaire d'une application.

10.3.1 Quiz

10.4 S4 : Programmation (Python)

Supports complémentaires :

- Diapositives: programmation (Python)
- Vidéo sur la programmation Python
- Un programme Python de lecture des données
- Une transaction Python

Voici maintenant quelques exemples de programmes accédant à notre base de données. Nous reprenons notre hypothèse d'une base nommée “”Messagerie », gérée par un SGBD relationnel (disons, ici, MySQL). Notre utilisatrice est Athénaïs : elle va écrire quelques scripts Python pour exécuter ses requêtes (Fig. 10.5).

Note : Le choix de Python est principalement motivé par la concision et la simplicité. On trouverait à peu près l'équivalent des programmes ci-dessous dans n'importe quel langage (notamment en Java,avec l'API JDBC). Par ailleurs, l'interface Python illustrée ici est standard pour tous les SGBD et nos scripts fonctionneraient sans doute à peu de chose près avec Postgres ou un autre.

Nos scripts sont des programmes *clients*, qui peuvent s'exécuter sur une machine, se connecter par le réseau au serveur de données, auquel ils transmettent des commandes (principalement des requêtes SQL). Nous sommes dans l'architecture très classique de la Fig. 10.5.

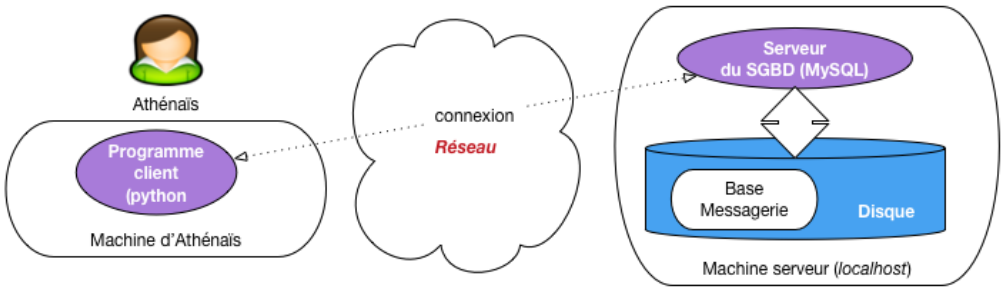

Fig. 10.5 – Architecture d'un programme dialoguant avec un serveur

10.4.1 Un programme de lecture

Pour établir une connexion, tout programme client doit fournir au moins 4 paramètres : l'adresse de la machine serveur (une adresse IP, ou le nom de la machine), le nom et le mot de passe de l'utilisateur qui se connecte, et le nom de la base. On fournit souvent également des options qui règlent certains détails de communication entre le client et le serveur. Voici donc la connexion à MySQL avec notre programme Python.

```
connexion = pymysql.connect
        ('localhost',
         'athénaïs',
         'motdepasse',
         'Messagerie',
          cursorclass=pymysql.cursors.DictCursor)
```

Ici, on se connecte à la machine locale sous le compte d'Athénaïs, et on accède à la base Messagerie. Le dernier paramètre est une option `cursorClass` qui indique que les données (nuplets) retournés par le serveur seront représentés par des dictionnaires Python.

Note : Un dictionnaire est une structure qui associe des clés (les noms des attributs) et des valeurs. Cette structure est bien adaptée à la représentation des nuplets.

Un curseur est créé simplement de la manière suivante :

```
curseur = connexion.cursor()
```

Une fois que l'on a créé un curseur, on s'en sert pour exécuter une requête.

```
curseur.execute("select * from Contact")
```

À ce stade, rien n'est récupéré côté client. Le serveur a reçu la requête, a créé le plan d'exécution et se tient prêt à fournir des données au client dès que ce dernier les demandera. Comme nous l'avons vu dans le chapitre sur la programmation, un curseur permet de parcourir le résultat d'une requête. Ici ce résultat est obtenu globalement avec la commande `fetchAll()` (on pourrait utiliser `fetchOne()`) pour récupérer les nuplets un par un). Le code Python pour parcourir tout le résultat est donc :

```
for contact in curseur.fetchall():
   print(contact['prénom'], contact['nom'])
```

La boucle affecte, à chaque itération, le nuplet courant à la variable `contact`. Cette dernière est donc un dictionnaire dont chaque entrée associe le nom de l'attribut et sa valeur.

Et voilà. Pour résumer, voici le programme complet, qui est donc remarquablement concis.

```
import pymysql
import pymysql.cursors

connexion = pymysql.connect('localhost', 'athénaïs',
```

(suite sur la page suivante)

(suite de la page précédente)

```
                    'motdepasse', 'Messagerie',
                    cursorclass=pymysql.cursors.DictCursor)

curseur = connexion.cursor()
curseur.execute("select * from Contact")

for contact in curseur.fetchall():
    print(contact['prénom'], contact['nom'])
```

Bien entendu, il faudrait ajouter un petit travail d'ingénierie pour ne pas donner les paramètres de connexion sous forme de constante mais les récupérer dans un fichier de configuration, et ajouter le traitement des erreurs (traiter par exemple un refus de connexion).

10.4.2 Une transaction

Notre second exemple montre une transaction qui sélectionne tous les messages non encore envoyés, les envoie, et marque ces messages en leur affectant la date d'envoi. Voici le programme complet, suivi de quelques commentaires.

```
1   import pymysql
2   import pymysql.cursors
3   from datetime import datetime
4
5   connexion = pymysql.connect('localhost', 'athénaïs',
6                   'motdepasse', 'Messagerie',
7                   cursorclass=pymysql.cursors.DictCursor)
8
9   # Tous les messages non envoyés
10  messages = connexion.cursor()
11  messages.execute("select * from Message where dateEnvoi is null")
12  for message in messages.fetchall():
13      # Marquage du message
14      connexion.begin()
15      maj = connexion.cursor()
16      maj.execute ("Update Message set dateEnvoi='2018-12-31' "
17          + "where idMessage=%s", message['idMessage'])
18
19      # Ici on envoie les messages à tous les destinataires
20      envois = connexion.cursor()
21      envois.execute("select * from Envoi as e, Contact as c "
22             +" where e.idDestinataire=c.idContact "
23             + "and  e.idMessage = %s", message['idMessage'])
24      for envoi in envois.fetchall():
25          mail (envoi['email'], message['contenu')
26
```

(suite sur la page suivante)

(suite de la page précédente)

```
27          connexion.commit()
```

Donc, ce programme effectue une boucle sur tous les messages qui n'ont pas de date d'envoi (lignes 10-12). À chaque itération, le cursor affecte une variable `message`.

Chaque passage de la boucle donne lieu à une transaction, initiée avec `connexion.begin()` et conclue avec `connexion.commit()`. Cette transaction effectue en tout et pour tout une seule mise à jour, celle affectant la date d'envoi au message (il faudrait bien entendu trouver la date du jour, et ne pas la mettre « en dur »).

Dans la requête `update` (lignes 16-17), notez qu'on a séparé la requête SQL et ses paramètres (ici, l'identifiant du message). Cela évite de construire la requête comme une chaîne de caractères. On ouvre ensuite un second curseur (lignes 20-24), sur les destinataires du message, et on envoie ce dernier.

Une remarque importante : les données traitées (message et destinataires) pourraient être récupérées en une seule requête SQL par une jointure. Mais le format du résultat (une table dans laquelle le message est répété avec chaque destinataire) ne convient pas du tout à la structure du programme dont la logique consiste à récupérer d'abord le message, puis à parcourir les envois, en deux requêtes. En d'autres termes, dans ce type de programme (très courant), SQL est sous-utilisé. Nous revenons sur ce point dans la dernière session.

10.4.3 Quiz

10.5 S5 : aspects transactionnels

Supports complémentaires :

- Diapositives: transactions
- Vidéo sur les transactions

Reprenons le programme transactionnel d'envoi de message. Même sur un exemple aussi simple, il est utile de se poser quelques questions sur ses propriétés dans un environnement sujet aux pannes et à la concurrence.

Une exécution de ce programme crée une transaction par message. Chaque transaction lit un message sans date d'envoi dans le curseur, envoie le message, puis modifie le message dans la base en affectant la date d'envoi. La transaction se termine par un `commit`. Que peut-on en déduire, en supposant un environnement idéal sans panne, où chaque transaction est la seule à accéder à la base quand elle s'exécute ? Dans un tel cas, il est facile de voir que *chaque message serait envoyé exactement une fois*. Les choses sont moins plaisantes en pratique, regardons-y de plus près.

10.5.1 Cas d'une panne

Imaginons (pire scénario) une panne *juste avant* le `commit`, comme illustré sur la Fig. 10.6. Cette figure montre la phase d'exécution, suivie de la séquence des transactions au sein desquelles on a mis en valeur celle affectant le message M_1.

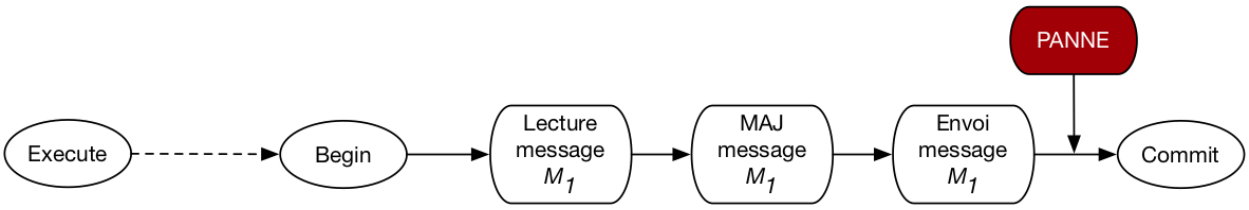

Fig. 10.6 – Cas d'une panne en cours de transaction

Au moment du redémarrage après la panne, le SGBD va effectuer un `rollback` qui affecte la transaction en cours. Le message reprendra donc son statut initial, sans date d'envoi. Il a pourtant été envoyé : l'envoi n'étant pas une opération de base de données, le SGBD n'a aucun moyen de l'annuler (ni même d'ailleurs de savoir quelle action le programme client a effectuée). C'est donc un premier cas qui viole le comportement attendu (chaque message envoyé exactement une fois).

Il faudra relancer le programme en espérant qu'il se déroule sans panne. Cette seconde exécution ne sélectionnera pas les messages traités par la première exécution *avant* M_1 puisque ceux-là ont fait l'objet d'une transaction réussie. Selon le principe de durabilité, le `commit` de ces transactions réussies n'est pas affecté par la panne.

10.5.2 Le curseur est-il impacté par une mise à jour ?

Passons maintenant aux problèmes potentiels liés à la concurrence. Supposons, dans un premier scénario, qu'une mise à jour du message M_1 soit effectuée par une autre transaction entre l'exécution de la requête et le traitement de M_1. La Fig. 10.7 montre l'exécution concurrente de deux exécutions du programme d'envoi : la première transaction (en vert) modifie le message et effectue un `commit` *avant* la lecture de ce message par la seconde (en orange).

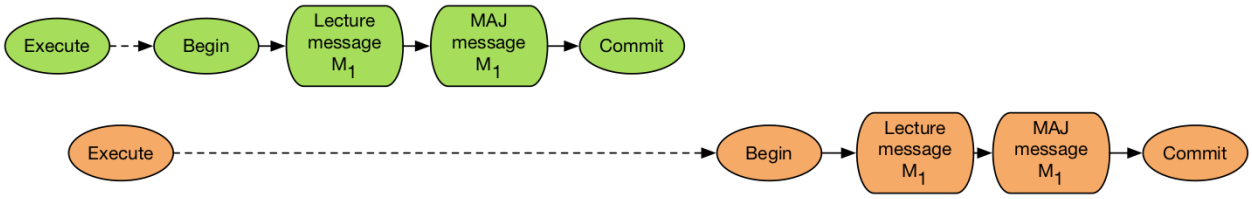

Fig. 10.7 – Cas d'une mise à jour *après* exécution de la requête mais *avant* traitement du message

Question : cette mise à jour sera-t-elle constatée par la lecture de M_1 ? Autrement dit, est-il possible que l'on constate, au moment de lire ce message dans la transaction orange, qu'il a déjà une date d'envoi parce qu'il a été modifié par la transaction verte ?

On pourrait être tenté de dire « Oui » puisqu'au moment où la transaction orange débute, le message a été modifié *et* validé. Mais cela voudrait dire qu'un curseur permet d'accéder à des données qui ne correspondent

pas au critère de sélection ! (En l'occurrence, on s'attend à ne recevoir que des messages sans date d'envoi). Ce serait très incohérent.

En fait, tout se passe comme si le résultat du curseur était un « cliché » pris au moment de l'exécution, et immuable durant tout la durée de vie du curseur. En d'autres termes, même si le parcours du résultat prend 1 heure, et qu'entretemps tous les messages ont été modifiés ou détruits, le système continuera à fournir *via* le curseur l'image de la base telle qu'elle était au moment de l'exécution.

En revanche, si on exécutait à nouveau une requête pour lire le message juste avant la modification de ce dernier, on verrait bien la mise à jour effectuée par la transaction verte. En résumé : une requête fournit la version des nuplets effective, soit au moment où la requête est exécutée (niveau d'isolation `read committed`), soit au moment où la transaction débute (niveau d'isolation `repeatable read`).

Conséquence : sur le scénario illustré par la Fig. 10.7, on enverra le message deux fois. Une manière d'éviter ce scénario serait de verrouiller tous les nuplets sélectionnés au moment de l'exécution, et d'effectuer l'ensemble des mises à jour en une seule transaction.

10.5.3 Transactions simultanées

Voici un dernier scénario, montrant un exécution simultanée ou quasi-simultanée de deux transactions concurrentes affectant le même message (Fig. 10.8).

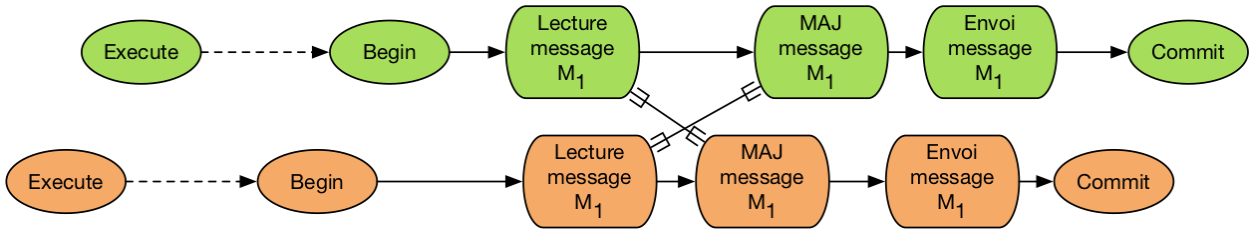

Fig. 10.8 – Exécution concurrente, avec risque de *deadlock*

Cette situation est très peu probable, mais pas impossible. Elle correspond au cas-type dit « des mises à jour perdues » étudié dans le chapitre sur les transactions. Dans tous les niveaux d'isolation sauf `serializable`, le déroulé sera le suivant :

— chaque transaction lit séparément le message
— une des transactions, disons la verte, effectue une mise à jour
— la seconde transaction (orange) tente d'effectuer la mise à jour et est mise en attente ;
— la transaction verte finit par effectuer un `commit`, ce qui libère la transaction orange : le message est envoyé deux fois.

En revanche, en mode `serializable`, chaque transaction va bloquer l'autre sur le scénario de la Fig. 10.8. Le système va détecter cet interblocage et rejeter une des transactions.

10.5.4 La bonne méthode

Ce qui précède mène à proposer une version plus sûre d'un programme d'envoi.

```
1  # Tous les messages non envoyés
2  messages = connexion.cursor()
3  messages.execute("SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE")
4
5  # Début de la transaction
6  connexion.begin()
7  messages.execute("select * from Message where dateEnvoi is null")
8
9  for message in messages.fetchall():
10     # Marquage du message
11     maj = connexion.cursor()
12     maj.execute ("Update Message set dateEnvoi='2018-12-31' "
13             + "where idMessage=%s", message['idMessage'])
14
15     print ("Envoi du message ...", message['contenu'])
16
17 connexion.commit()
```

Tout d'abord (ligne 3) on se place en niveau d'isolation sérialisable.

Puis (ligne 5), on débute la transaction à *l'extérieur* de la boucle du curseur, et on la termine après la boucle (ligne 17). Cela permet de traiter la requête du curseur comme partie intégrante de la transaction.

Au moment de l'exécution du curseur, les nuplets sont réservés, et une exécution simultanée sera mise en attente si elle essaie de traiter les mêmes messages.

Avec cette nouvelle version, la seule cause d'envoi multiple d'un message et l'occurence d'un panne. Et le problème dans ce cas vient du fait que l'envoi n'est pas une opération contrôlée par le serveur de données.

10.5.5 Quiz

10.6 S6 : *mapping* objet-relationnel

Supports complémentaires :

— Diapositives: Mapping objet-relationnel
— Vidéo sur le mapping objet-relationnel

Pour conclure ce cours, voici une discussion sur la méthodologie d'association entre une base de données relationnelle et un langage de programmation, en supposant de plus que ce langage est orienté-objet (ce qui est très courant). Nous avons montré comment intégrer des requêtes SQL dans un langage de programmation, Java (chapitre *Procédures et déclencheurs*), Python (le présent chapitre), et les mêmes principes s'appliquent à PHP, C#, C++, ou tout autre langage, objet ou non.

Cette intégration est simple à réaliser mais assez peu satisfaisante en terme d'ingénierie logicielle. Commençons par expliquer pourquoi avant de montrer des environnements de développement qui visent à éviter le problème, en associant objets et relations, en anglais *object-relationnal mapping* ou ORM.

10.6.1 Quel problème

Le problème est celui de la grande différence enttre deux représentations (deux modèles) des données

- dans un langage objet, les données sont sous forme d'objets, autrement dit des petits systèmes autonomes dotés de propriétés (les données) dont la structure est parfois complexe, étroitement liées à un comportement (les méthodes)
- dans une base relationnelle, les données sont des nuplets, de structure élémentaire (un dictionnaire associant des noms et des valeurs atomiques), sans aucun comportement.
- dans un langage objet, les objets sont liés les uns aux autres par un référencement physique (pointeurs ou équivalent), et une application manipule donc un *graphe d'objets*
- dans une base relationnelle, les nuplets sont liés par un mécanisme « logiaue » de partage de valeurs (clé primaire, clé étrangère) et on manipule des ensembles, pas des graphes.

Le problème d'une intégration entre un langage de programmation est SQL est donc celui de la *conversion* d'un modèle à l'autre. Ce n'était pas flagrant sur les quelques exemples simples que nous avons donnés, mais à l'échelle d'une application d'envergure, cette conversion devient pénible à coder, elle ne présente aucun intérêt applicatif, et entraine une perte de productivité peu satisfaisante.

Note : Notons au passage que pour éviter ces écarts entre modèles de données, on a beaucoup travaillé pendant une période sur les bases objets et pas relationnelles. Cette recherche n'a pas vraiment abouti à des résultats vraiment satisfaisants.

Voici un exemple un peu plus réaliste que ceux présentés jusqu'à présent pour notre application de messagerie. Dans une approche objet, on modéliserait nos données par des classes, soit une classe `Contact` et une classe `Message`. Voici pour commencer la classe `Contact`, très simple : elle ne contient que des propriétés et une méthode d'initialisation.

```
class Contact:
  def __init__(self,id,prenom,nom,email):
              self.id=id
              self.prenom=prenom
              self.nom=nom
              self.email=email
```

Et voici comment on effectue la conversion : dans une boucle sur un curseur récupérant des contacts, on construit un objet de la classe `Contact` en lui passant comme valeurs d'initialisation celles provenant du curseur.

```
curseur.execute("select * from Contact")
for cdict in curseur.fetchall():
   # Conversion du dictionnaire en objet
   cobj = Contact(cdict["idContact"], cdict["prénom"],
              cdict["nom"], cdict["email"])
```

C'est la conversion la plus simple possible : elle ne prend qu'une instruction de programmation. C'est déjà trop dans une optique de productivité optimale : on aimerait que le curseur nous donne *directement* l'objet instance de `Contact`.

Les choses se gâtent avec la classe `Message` dont la structure est beaucoup plus complexe. Voici tout d'abord sa modélisation Python.

```
class Message:
    # Emetteur: un objet 'Contact'
   emetteur = Contact(0, "", "", "")
   # Prédecesseur: peut ne pas exister
   predecesseur = None
   # Liste des destinataires = des objets 'Contacts'
   destinataires = []

   # La méthode d'envoi de message
   def envoi(self):
     for dest in self.destinaires:
```

L'envoi d'un message prend naturellement la forme d'une méthode de la classe. Envoyer un message devient alors aussi simple que l'instruction :

```
message.envoi()
```

Un message est un nœud dans un graphe d'objet, il est lié à un objet de la classe `Contact` (l'émetteur), et aux destinataires (objets de la classe `Contact` également). Pour instancier un objet de la classe `Message` à partir de données provenant de la base, il faut donc :

— Lire le message et le convertir en objet
— Lire l'émetteur et le convertir en objet de la classe `Contact`
— Lire tous les destinataires et les convertir en objets de la classe `Contact`

Cela donne beaucoup de code (je vous laisse essayer si vous les souhaitez), d'un intérêt applicatif nul. De plus, il faudrait idéalement qu'à chaque nuplet d'une table corresponde un seul objet dans l'application. Avant d'instancier un objet `Contact` comme émetteur, il faudrait vérifier s'il n'a pas déjà été instancié au préalable et le réutiliser. On aurait ainsi, pour cet objet, un lien inverse cohérent : la liste des messages qu'il a émis. En fait, pour l'application objet, les données ont la forme que nous avons déjà illustrées par la Fig. 10.9.

Bref, il faudrait que le graphe soit une image cohérente de la base, conforme à ce qui illustré par la figure, et ce n'est pas du tout facile à faire.

10.6.2 Quelle solution

Le rôle d'un système ORM est de convertir automatiquement, à la demande, la base de données sous forme d'un graphe d'objet. L'ORM s'appuie pour cela sur une configuration associant les classes du modèle fonctionnel et le schéma de la base de données. L'ORM génère des requêtes SQL qui permettent de matérialiser ce graphe ou une partie de ce graphe en fonction des besoins.

La Fig. 10.10 illustre l'architecture d'un système ORM. Il présente à l'application les données sous la forme d'une graphe d'objets (en haut de la figure). Ce graphe est obtenu par production automatique de requêtes SQL et conversion du résultat de ces requêtes en objets.

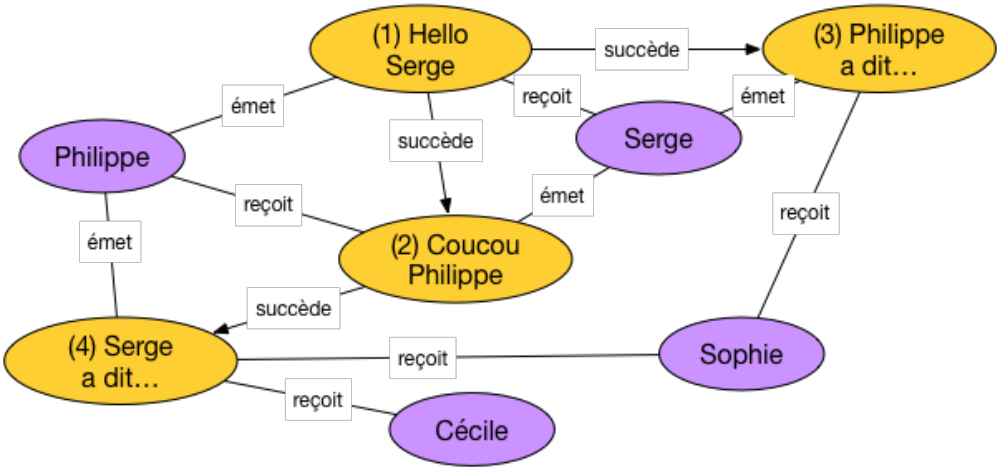

Fig. 10.9 – Une instance (petite mais représentative) de notre messagerie

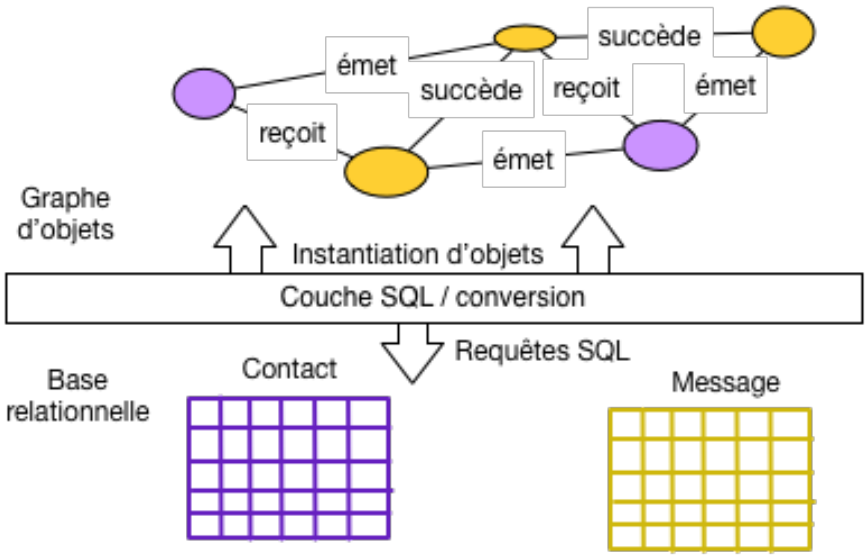

Fig. 10.10 – Architecture d'un système ORM

Un système ORM s'appuie sur une configuration qui décrit la correspondance (le *mapping*) entre une classe et une table. Voici par exemple cette spécification pour un des systèmes ORM les plus développés, Hibernate.

```
@Entity(table="Message")
public class Message {

   @Id
   private Integer id;

   @Column
   private String contenu;

   @ManyToOne
   private Contact emetteur;

   @OneToMany
   private Set<Contact> destinataires ;
 }
```

Les annotation `@Entity`, `@Id`, `@Column`, `@ManyToOne`,``@OneToMany``, indiquent au système ORM tout ce qui est nécessaire pour associer objets et nuplets de la base. Ce même système est alors en mesure de produire les requêtes SQL et de les soumettre au SGBD via l'interface JDBC, ou l'API Python.

Le gain en terme de productivité est très important. Voici, toujours en Hibernate (la syntaxe est la plus claire).

```
List<Message> resultat =
     session.execute("from Message as m "
                   + " where m.emetteur.prenom = 'Serge'");

for (Message m : resultat) {
    for (Contact c : m.destinataires) {
        message.envoi (c.email);
    }
}
```

Notez les deux boucles, la première sur le messages, la seconde sur leurs destinataires. Dans le second cas, aucune requête n'a été executée explicitement : c'est le système ORM qui s'est chargé automatiquement de trouver les destinataires du messages et de les présenter sous la forme d'instances de la classe `` Contact``.

En résumé, les systèmes ORM sont maintenant très utilisés pour les développements d'envergure. Leurs principes sont tous identiques : les accès à la base prennent la forme d'une navigation dans un graphe d'objets et le système engendre les requêtes SQL pour matérialiser le graphe. La conversion entre nuplets de la base et objets de l'application est *automatique*, ce qui représente un très important gain en productivité de développement. En contrepartie, tend à produire *beaucoup* de requêtes élémentaires là où *une seule* jointure serait plus efficace. Pour des bases très volumineuses, l'intervention d'un expert est souvent nécessaire afin de contrôler les requêtes engendrées.

Voici pour cette brève introduction. Pour aller plus loin, l'atelier ci-dessous propose un début de développement avec le *framework* Django. Vous pouvez aussi consulter le cours complet http://orm.bdpedia.fr consacré à Hibernate.

10.6.3 Quiz

10.7 Atelier : une application Django

Dans cet atelier nous allons débuter la mise en place d'une application de messagerie avec le *framework* Django, un environnement complet de développement Python, essentiellement orienté vers le développement d'applications Web. Django comprend un composant ORM sur lequel nous allons bien sûr nous concentrer.

Note : Cet atelier suppose quelques compétences en Python et en programmation objet. Si ce n'est pas votre cas, contentez-vous de lire (et comprendre) les différentes étapes détaillées ci-dessous.

Supports complémentaires :

— Le code de l'application Django

10.7.1 Préliminaires

Pour cet atelier vous avez besoin d'un serveur de données relationnel et d'un environnement Python 3. Il faut de plus installer Django, ce qui est aussi simple que :

```
pip3 install django
```

Vérifier que Django est bien installé.

```
python3

  >>> import django
  >>> print(django.__path__)
  >>> print(django.get_version())
```

Django est installé avec un utilitaire `django-admin` qui permet de créer un nouveau projet. Supposons que ce projet s'appelle `monappli`.

```
django-admin startproject monappli
```

Django crée un répertoire `monappli` avec le code initial du projet. La Fig. 10.11 montre les quelques fichiers et répertoires créés.

Le projet est déjà opérationnel : vous pouvez lancer un petit serveur web :

```
cd monappli
python3 manage.py runserver
```

Le serveur est en écoute sur le port 8000 de votre machine. Vous pouvez accéder à l'URL http://localhost:8000 avec votre navigateur. Vous devriez obtenir l'affichage de la Fig. 10.12.

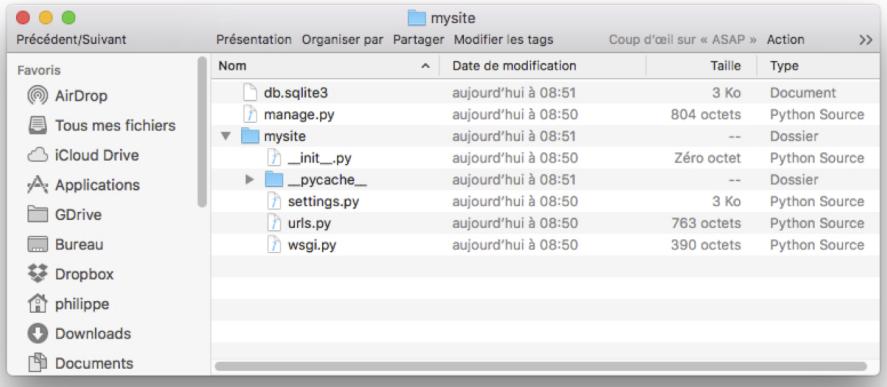

Fig. 10.11 – Le squelettte du projet `monappli`, créé par Django

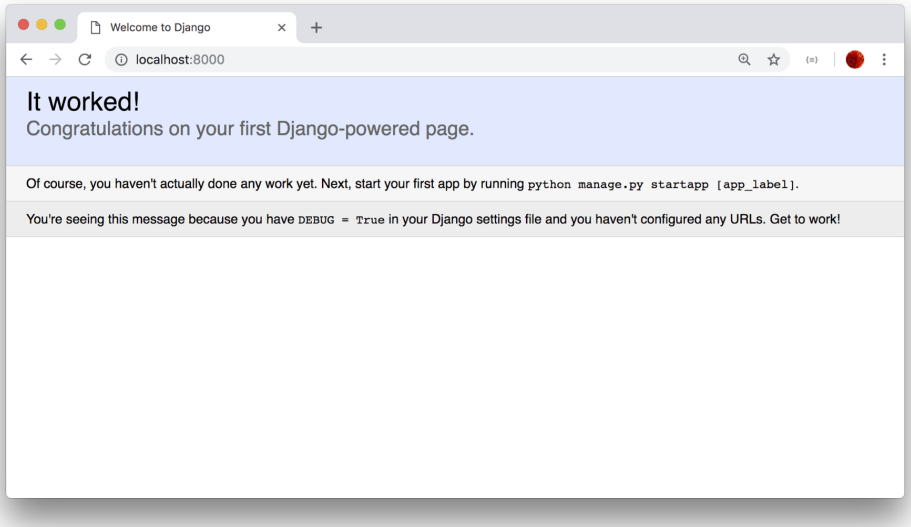

Fig. 10.12 – La première page de l'application

Evidemment pour l'instant il n'y a pas grand chose. À partir de là, on peut travailler sur le projet avec un éditeur de texte ou (mieux) un IDE comme Eclipse.

10.7.2 L'application `messagerie`

Un projet Django est un ensemble d'applications. Au départ, une première application est créée et nommée comme le projet (donc, pour nous, `monappli`, cf. Fig. 10.11). Elle contient la configuration. Créons notre première application avec la commande suivante :

```
python3 manage.py startapp messagerie
```

Django ajoute un répertoire `messagerie` qui vient s'ajouter à celui de l'application de configuration nommé `monappli`. On obtient le contenu du projet illustré par la Fig. 10.13.

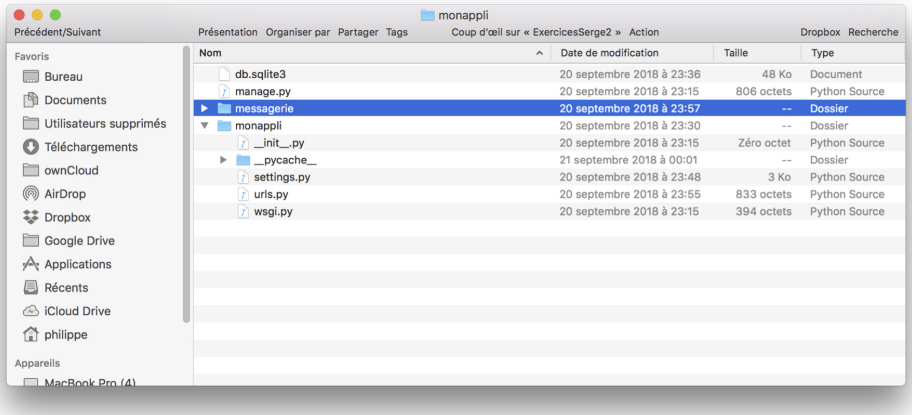

Fig. 10.13 – Nouvelle application

Maintenant, il faut ajouter l'application `messagerie` dans la configuration du projet, contenue dans le fichier `monappli/monappli/settings.py`. Editez ce fichier et modifier le tableau `INSTALLED_APPS` comme suit.

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    "messagerie"
]
```

Tant que nous y sommes, configurons la base de données. Supposons que le système soit MySQL, sur la machine locale, avec la base et l'utilisateur suivant :

```
create database Monappli;
grant all on Monappli.* to philippe identified by 'motdepasse'
```

Toujours dans le fichier `monappli/monappli/settings.py`, reportez ces paramètres dans `DATABASES` :

```
DATABASES = {
   'default': {
       'ENGINE': 'django.db.backends.mysql',
       'NAME': 'Monappli',
       'USER': 'philippe',
       'PASSWORD': 'motdepasse',
       'HOST': '127.0.0.1',
       'PORT': '3306',
    }
}
```

Bien sûr, vous pouvez utiliser un autre système ou d'autres valeurs de paramètres. Voilà ! Nous sommes prêts à utiliser la couche ORM.

10.7.3 L'ORM de Django : le schéma

Dans Django, toutes les données sont des objets Python. Quand ces données sont *persistantes*, on déclare les classes comme des *modèles*, et Django se charge alors de convertir automatiquement les objets de ces classes en nuplets de la base de données.

Note : Pourquoi « modèle » ? Parce que Django est basé sur une architecture dite MVC, pour Modèle-Vue-Contrôleur. Les « modèles » dans cette architecture sont les données persistantes.

Django (son composant ORM) se charge également automatiquement de créer et modifier le schéma de la base. À chaque ajout, modification, suppression d'une classe-modèle, la commande suivante crée des commandes SQL d'évolution du schéma (nommée *migration* dans Django).

```
python3 manage.py makemigrations messagerie
```

Les commandes de migration sont stockées dans des fichiers placés dans un sous-répertoire `migrations`. On applique alors ces commandes SQL en exécutant :

```
python3 manage.py migrate
```

Allons-y. Nous allons créer une première classe `Contact`, que nous plaçons dans le fichier `monappli/messagerie/models.py`.

```
from django.db import models

class Contact(models.Model):
    email = models.CharField(max_length=200)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    prenom = models.CharField(max_length=200)
    nom = models.CharField(max_length=200)

    def __str__(self):
        return self.prenom + ' ' + self.nom

    class Meta:
        db_table = 'Contact'
```

Tous les attributs persistants (ceux qui vont être stockées dans la base) doivent être d'un type fourni par les modèles Django, ici `CharField` qui correspond à `varchar`. Notez que l'on ne définit pas de clé primaire : un objet Python a son propre identifiant, qu'il est inutile de déclarer. En revanche, Django va créer automatiquement un identifiant de base de données. Notez également que l'on peut spécifier le nom de la table associée dans les méta-données de la classe. Sinon, Django fournira un nom par défaut (c'est le cas pour les attributs, comme nous allons le voir).

Nous avons donc une classe Python classique, enrichie des spécifications permettant de faire le lien avec la base de données. Nous pouvons alors demander à Django de créer la table correspondante.

```
python3 manage.py makemigrations messagerie
```

Vous devriez obtenir la réponse suivante :

```
Migrations for 'messagerie':
    messagerie/migrations/0001_initial.py:
- Create model Contact
```

Explication : Django a détecté une nouvelle classe-modèle `Contact`, et a créé la commande SQL correspondante dans le fichier `messagerie/migrations/0001_initial.py`. Pour contrôler cette requête vous pouvez demander son affichage :

```
python3  manage.py sqlmigrate messagerie 0001
```

Ce qui donne l'affichage suivant :

```
BEGIN;
--
-- Create model Contact
--
CREATE TABLE `Contact` (`id` integer AUTO_INCREMENT NOT NULL PRIMARY KEY,
                        `email` varchar(200) NOT NULL,
                        `prenom` varchar(200) NOT NULL,
                        `nom` varchar(200) NOT NULL);
COMMIT;
```

Vous pouvez apprécier comment les définitions de la classe objet Python ont été transcrites en commande SQL. En particulier, une clé primaire a été ajoutée, un entier auto-incrémenté. Il reste à appliquer cette commande de création :

```
python3 manage.py migrate
```

À ce stade, vous devriez avoir la table `Contact` créée dans votre schéma `Monappli` : vérifiez ! C'est peut-être le moment de faire une pause pour bien assimiler ce que nous venons de faire. En résumé : Django effectue maintenant la correspondance entre la classe Python `Contact` et la table `Contact` dans MySQL.

Avant de vérifier comment nous pouvons exploiter cette correspondance, passons à la table `Message` qui illustre la gestion des liens entre objets (en Python).

```
class Message(models.Model):
    emetteur = models.ForeignKey(Contact, on_delete=models.CASCADE,
                                 related_name='messages_emis')
    contenu = models.TextField()
    date_envoi = models.DateTimeField()
    predecesseur = models.ForeignKey('self', on_delete=models.CASCADE,
                         null=True, blank=True, related_name='successeurs')

    class Meta:
        db_table = 'Message'

    def __str__(self):
        return self.contenu
```

Les attributs `emetteur` et `predecesseur` sont de type `ForeignKey` et désignent respectivement un objet de la classe `Contact` et un objet de la classe `Message` (lien réflexif donc, désigné par `self`). La valeur de `related_name` donne le nom d'une association vu du côté opposé à celui où elle est définie. Du côté de la classe `Contact` par exemple, on peut accéder à tous les messages émis par un contact avec un attribut nommé `messages_emis`. Un exemple est donné plus loin.

Note : Django impose l'ajout d'une clause `on_delete` : voir le chapitre *Schémas relationnel* pour des explications à ce sujet.

Créez la spécification de la table avec la commande `makemigrations`, puis créez la table avec `migrate`, comme précédemment.

Voilà, nous savons créer et modifier un schéma.

Exercice Ex-django-1 : finalisez le schéma

Créez les classes `Contact` et `Message` comme indiqué ci-dessus.

Créez ensuite une classe `Envoi` pour compléter notre schéma.

Voyons maintenant (enfin) comment on gère les données.

10.7.4 L'ORM de Django : les données

Django a créé une interface Web d'administration des données ! Cette interface est accessible au super-utilisateur du projet, que nous pouvons créer avec la commande suivante :

```
python3 manage.py createsuperuser
```

Entrez un compte super-utilisateur, à votre choix. Il reste à indiquer que l'on veut engendrer une interface sur nos classes-modèles. Ajoutez pour cela les lignes suivantes dans `messagerie/admin.py`.

```
from django.contrib import admin

from .models import Message, Contact

admin.site.register(Contact)
admin.site.register(Message)
```

L'interface d'administration est automatiquement engendrée et gérée par le framework. Elle est accessible à l'URL http://localhost:8000/admin/. Django vous demande au préalable de vous connecter avec le compte super-utilisateur créé précédemment : vous devriez alors obtenir l'affichage de la Fig. 10.14.

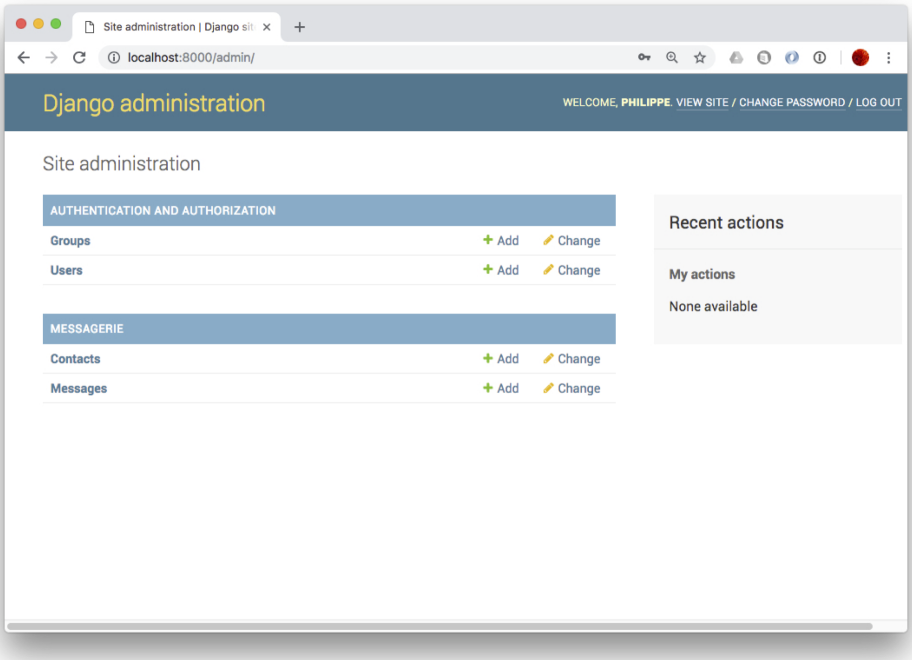

Fig. 10.14 – L'interface d'administration de Django

À vous de jouer : accédez à l'interface sur les contacts, explorez-là, appréciez comment, à partir d'une simple correspondance entre la classe objet et la base, il est possible d'engendrer une application permettant les opérations dites CRUD (*Create, Read, Update, Delete*).

Exercice Ex-django-2 : insérez les données

Utilisez l'interface d'administration pour créez l'instance de la Fig. 10.4.

10.7.5 Les vues

Et pour finir nous allons utiliser la couche ORM de Django pour naviguer dans la base de données sans écrire une seule requête SQL (Django s'en charge tout seul). Dans un modèle MVC, la notion de « vue » correspond à la couche de présentation des données. Une vue comprend deux parties :

— une *action*, implantée en Python, qui effectue par exemple des accès à la base de données et/ou des traitements
— un modèle d'affichage (*template*), à ne pas confondre avec de modèle de données.

Chaque vue est associée à une adresse Web (une URL). La correspondance entre les vues et leur URL est indiquée dans des fichiers `urls.py`. Commencez par éditer le fichier `monappli/urls.py` comme suit :

```
from django.urls import path, include
from django.contrib import admin


urlpatterns = [
    path('admin/', admin.site.urls),
    path('messagerie/', include('messagerie.urls')),
]
```

Puis, éditez le fichier `monappli/messagerie/urls.py`

```
from . import views
from django.urls import path

urlpatterns = [
    path('contacts', views.contacts, name='contacts'),
]
```

Cette configuration nous dit que l'URL http://localhost:8000/messagerie/contacts correspond à la fonction `contacts()` des vues de l'application `messagerie`. Ces fonctions sont toujours placées dans le fichier `messagerie/views.py`. Placez-y le code suivant pour la fonction `contacts()`.

```
from django.shortcuts import render
from .models import Contact, Message

def contacts(request):
    messages = Contact.objects.all()
    context = {'les_contacts': contacts}

    return render(request, 'messagerie/contacts.html', context)
```

Que fait cette fonction ? Elle commence par « charger » tous les messages avec l'instruction :

```
messages = Message.objects.all()
```

C'est ici que la couche ORM intervient : cette appel déclenche la requête SQL suivante

```
select * from Contact
```

Les nuplets obtenus sont alors *automatiquement* convertis en objets de la classe `Contact`. Ces objets sont ensuite transmis au modèle d'affichage (le « *template* ») que voici (à sauvegarder dans `monappli/messagerie/templates/messagerie/contacts.html`).

```
{% if les_contacts %}
    <ul>
   {% for contact in les_contacts %}
        <li>
              {{ contact.prenom }}   {{ contact.nom }}
      </li>
   {% endfor %}
   </ul>
{% else %}
    <p>Aucun contact.</p>
{% endif %}
```

Voilà, oui, je sais, il y a beaucoup de choses qui peuvent sembler obscures. Cela mérite une lecture soigneuse suivie d'autant de relectures que nécessaire. En tout cas une fois tout cela complété, vous devriez pouvoir accéder à l'URL http://localhost:8000/messagerie/contacts et obtenir l'affichage des contacts que vous avez entrés, comme sur la Fig. 10.15.

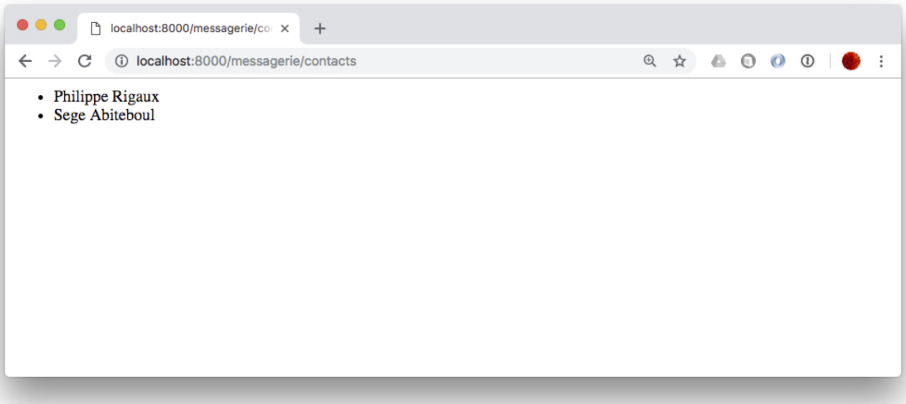

Fig. 10.15 – La liste des contacts, affichés par notre vue

On peut faire la même chose avec les messages bien entendu. Le fichier `urls.py` est le suivant :

```
from . import views
from django.urls import path

urlpatterns = [
    path('contacts', views.contacts, name='contacts'),
    path('messages', views.messages, name='messages'),
]
```

La fonction suivante est ajoutée dans `views.py` :

```
def messages(request):
  messages = Message.objects.all()
  context = {'les_messages': messages}

  return render(request, 'messagerie/messages.html', context)
```

Et finalement, voici le *template* `monappli/messagerie/templates/messagerie/messages.html`.

```
{% if les_messages %}
    <ul>
   {% for message in les_messages %}
        <li>
         "{{message.contenu}}"" envoyé par
            {{ message.emetteur.prenom }}   {{ message.emetteur.nom }}
      </li>
   {% endfor %}
   </ul>
{% else %}
    <p>Aucun message.</p>
{% endif %}
```

Vous remarquez sans doute (ou peut-être) que nous pouvons accéder à l'émetteur du message avec l'expression `message.emetteur`. C'est un exemple de *navigation* dans un modèle objet (passage de l'objet `Message` à l'objet `Contact`) qui correspond à une jointure en SQL. Cette jointure est automatiquement effectuée par la couche ORM de Django. Plus besoin de faire (directement en tout cas) du SQL !

Exercice Ex-django-3 : affichez toutes les données

Affichez non seulement l'émetteur du message mais la liste des destinataires. Aide : dans la classe `Envoi` nommez l'association vers les destinataires avec `related_name`.

```
message = models.ForeignKey(Message, related_name='destinataires')
```

À partir du message, on obtient alors les destinataires avec l'expression `message.destinataires`. Pour parcourir les destinataires, on écrit donc quelque chose comme :

```
 {% for dest in message.destinataires.all %}
    # Affichage
{% endfor %}
```

Vous pouvez aussi créer une nouvelle vue qui affiche les contacts avec les messages qu'ils ont envoyés.

10.8 Atelier : votre étude de cas

Vous devriez maintenant être totalement autonome pour implanter une base de données d'un niveau de sophistication moyen. C'est ce que vous allez essayer de vérifier dans cet atelier.

10.8.1 La modélisation initiale

Notre étude porte sur la réalisation d'une application de gestion des treks. Qu'est-ce qu'un trek ? C'est une randonnée sur plusieurs jours, découpées en étapes allant d'un point d'hébergement (un refuge, un hôtel, un bivouac) à un autre. Chaque étape est elle-même constituée de sections décrites par un tracé GPS et rencontrant un ensemble de point d'intérêts. Enfin, un point d'intérêt est une localisation distinctive sur le parcours : un monument (fontaine, église, tombeau), un cours d'eau à traverser, un belvédère, etc.

Après quelques mois de réunions (parfois houleuses) consacrées à la conception de la base, l'équipe chargée du projet est arrivée au schéma E/A de la Fig. 10.16. Ce sera notre point de départ.

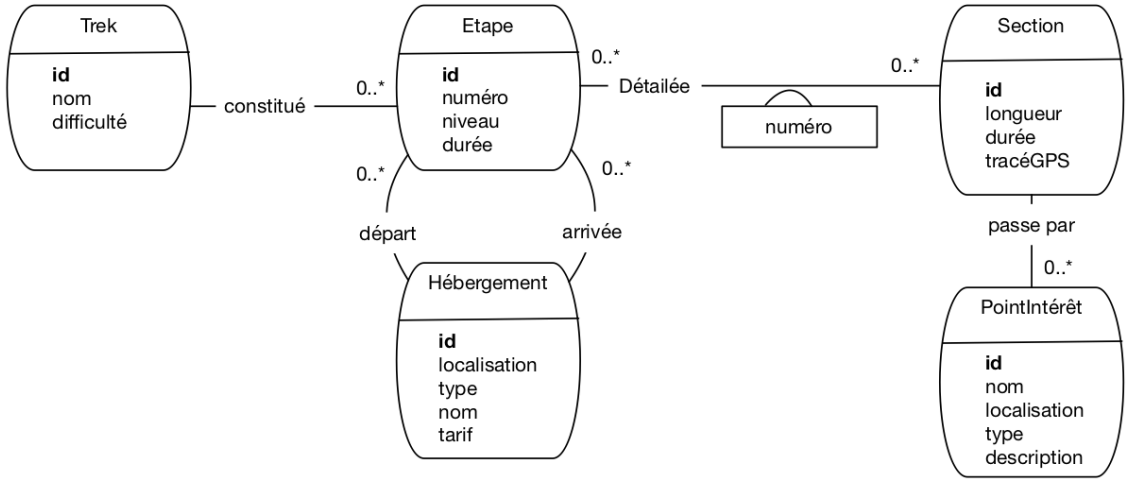

Fig. 10.16 – La modélisation initiale de notre base de treks

Voici quelques explications sur les choix effectués. Un trek est donc constitué d'étapes numérotées séquentiellement. Pour chaque étape on connaît l'hébergement de départ et celui d'arrivée. Notez que les concepteurs ont considéré qu'un même hébergement pouvait être le départ de plusieurs étapes, voire le départ d'une étape et l'arrivée d'une autre, ou même le départ et l'arrivée d'une même étape.

Une étape est donc une séquence de sections. Ici, il a finalement été décidé qu'une même section pouvait être partagée par des étapes distinctes (de treks eux-mêmes éventuellement distincts). La localisation est en

fait une paire d'attributs (longitude, lattitude) dont les valeurs sont exprimées en coordonnées GPS. Ca se fait en relationnel. Un problème plus délicat est celui du tracé GPS qui ne semble pas compatible avec une modélisation relationnelle. Les concepteurs ont donc imaginé le stocker dans une chaîne de caractères. Cette gestion de données géolocalisées n'est pas très satisfaisante, nous y reviendrons.

Finalement, sur le parcours d'une section on trouve des points d'intérêt.

Quelqu'un a fait remarquer que la durée d'une étape est sans doute la somme des durées des sections, et que cela constituait une redondance. Le chef de projet a considéré que ce n'était pas important.

Tout cela semble à peu près correct. C'est ce que nous appellerons la modélisation initiale. Maintenant à vous de jouer.

10.8.2 Peut-on faire mieux ?

Posons-nous des questions sur ce schéma pour savoir si d'autres choix sont possibles, et éventuellement meilleurs.

Entités faibles

Certains types d'entités vous semblent-ils de bon candidats pour être transformés en type d'entités faibles ? Proposez la solution correspondante, et donnez votre avis sur l'intérêt de l'évolution.

Associations réifiées

Même question, cette fois pour les associations réifiées.

Spécialisation (difficile)

Un œil exercé remarque que les types d'entités *Hébergement* et *PointIntérêt* partagent des attributs importants comme le nom, la localisation, le type. Et si un hébergement n'était qu'un point d'intérêt d'un type particulier, se demande cet expert ? On pourrait en profiter pour être plus précis sur les types de points d'intérêt et les attributs spécifiques qui les caractérisent. On pourrait par exemple détailler les points de ravitaillement avec leurs heures d'ouverture, les sommets avec leur altitude, les lacs avec leur superficie,... Que deviendrait alors la modélisation ? Exercice un peu difficile, mais très instructif !

Récursion (très difficile)

Le même expert (il coûte cher mais il en vaut la peine) pose la question suivante : existe-t-il vraiment une différence profonde de nature entre un trek, une étape et une section ? N'est-on pas en train de définir une hiérarchie à trois niveaux pour un même concept générique de *parcours* alors qu'un petit effort complémentaire permettrait de représenter une hiérarchie plus générique, sans limitation de niveau et sans multiplier des types d'entités proches les uns des autres. Argument fort : on pourrait à l'avenir étendre l'application à d'autres types de parcours : voies d'escalade, rallye automobiles, courses de vélo, etc.

Une telle extension de la modélisation passe par une combinaison de concepts avancés : spécialisation et récursion. À tenter quand tout le reste est maîtrisé. Vraiment difficile mais stimulant.

10.8.3 Une instance-échantillon

Comme pour l'application de messagerie, essayez de dessiner un graphe "instance montrant quelle structure aura votre base de données. Prenez un trek de votre choix, cherchez la description sur le Web et vérifiez si le schéma permet de représenter correctement les informations données sur les sites trouvés en étudiant 2 ou 3 étapes.

Si vous n'avez pas d'inspiration, voici une possibilité : deux treks, le GR5 et le Tour du Mont-Blanc (TMB) partagent une partie de leur parcours. Voici, avec une petite simplification, de quoi constituer votre échantillon.

— l'étape 1 du TMB va du chalet de La Balme au refuge Robert Blanc
— l'étape Alpes-4 du GR5 va des Contamines au Cormet de Roselend

Les sections concernées sont :

— Section 1 : des Contamines au chalet de la Balme
— Section 2 : du chalet de la Balme au Col du Bonhomme
— Section 3 : du Col du Bohomme au refuge des Mottets
— Section 4 : du Col du Bohomme au Cormet de Roselend

L'étape 1 du TMB est constituée des sections 2 et 3, l'étape Alpes-4 du GR5 des sections 1, 2 et 4.

Quelques points d'intérêts : sur la section 1, on passe par le chalet de Nant Bornant. sur la section 3, on passe par Ville des Glaciers. Notez que le chalet de la Balme est hébergement pour le TMB, un point d'intérêt pour le GR5, d'où l'intérêt sans doute d'unifier les deux concepts par une spécialisation. Sinon, il créer le chalet de la Balme comme un hébergement d'une part, un point d'intérêt de l'autre. Vous voilà confrontés aux soucis typiquesdu concepteur. Si vous ne vous sentez pas à l'aise, faites au plus simple.

Représentez le graphe de ces entités conformément au schéma de notre modélisation. Si vous avez la curiosité de regarder ces parcours sur un site web, vous trouverez certainement des données qui pourraient être ajoutée (le dénivellé positif et négatif par exemple). Demandez-vous où les faire figurer.

10.8.4 Le schéma relationnel

Passons maintenant au schéma relationnel. À partir de la modélisation initiale

— donnez la liste complète des dépendandes fonctionnelles définies par le schéma de la Fig. 10.16
— appliquez l'algorithme de normalisation pour obtenir le schéma relationnel.

Vous pouvez également produire les variantes issues des choix alternatifs de modélisation (entités faibles, réification) pour méditer sur l'impact de ces choix. Les plus courageux pourront produire la modélisation relationnelle correspondant à l'introduction d'une structure de spécialisation.

10.8.5 Les `create table`

Produisez les commandes de création de table, en indiquant soigneusement les clés primaires et étrangère.

Produisez également les commandes d'insertion pour l'instance échantillon.

Idéalement, vous disposez d'un serveur de données installé quelque part, et vous avez la possibilité de créer le schéma et l'instance.

10.8.6 Les vues

Vous souvenez-vous de la redondance entre la durée d'une étape et celle de ses sections (la première est en principe la somme des secondes). Cette redondance et source d'anomalies. Par exemple si je modifie la longueur d'une section suite à un changement d'itinéraire, je ne devrais pas oublier de reporter cette modification sur les étapes comprenant cette section.

On pourrait supprimer la durée au niveau de l'étape, mais, d'un autre côté, disposer de la longueur d'une étape est bien pratique pour l'interrogation. Il existe une solution qui met tout le monde d'accord : créer une vue qui va *calculer* la durée d'une étape et prendra donc automatiquement en compte tout changement sur celle des sections.

Créez cette vue !

10.8.7 Interrogation SQL

Il ne reste plus qu'à exprimer quelques requêtes SQL. Exemples (à adapter à votre instance échantillon) :

- Afficher les sections qui font partie du TMB
- Afficher les étapes avec les noms et tarifs de leurs hébergement de départ et d'arrivée
- Quels points d'intérêts sont partagés entre deux trek différents ?
- Quelles étapes ont plus de deux sections ?
- Paires d'étapes dont l'hébergement d'arrivée de la seconde est l'hébergement de départ de la seconde
- Quelles sections n'ont pas de point d'intérêt
- Etc.

Idéalement toujours, vous pouvez interroger directement votre instance-échantillon. Pour MySQL par exemple, l'utilitaire phpMyAdmin dispose d'une fenêtre pour entrer des requêtes SQL et les exécuter.

10.8.8 Et pour aller plus loin

Reprenons le cas des données géolocalisées. Les représenter en relationnel est très peu satisfaisant. C'est à peu près acceptable pour une localisation (on donne la lattitude et la longitude), mais pour un tracé, ou une surface, ça ne va pas du tout. Le problème est que ce type de données est associé à des opérations (distance entre deux points, calcul de longueurs ou d'intersection) qui ne sont pas exprimables en SQL.

Très tôt, l'idée d'enrichir le système de types de SQL avec des *types abstraits de données* (TAD) a été proposée. Les types géométriques sont des exemples typiques de TAD extrêmement utiles et ont été introduits, par exemple, dans Postgres.

À vous de creuser la question si cela vous intéresse. Vous pouvez regarder *PostGIS* (Postgres Geographic Information System) ou les extensions existant dans d'autres systèmes. Vous sortez des limites du cours et devenez un véritable expert en bases de données. Bonne route sur ce trek difficile mais exaltant !

CHAPITRE 11

Annales des examens

11.1 Examen blanc, janvier 2019

Le schéma de base de données suivant sera utilisé pour l'ensemble du sujet. Il permet de gérer une plateforme de vente de billets de spectacles en ligne :

- Personne(id, nom, prénom, age)
- Salle(id, nom, adresse, nbPlaces)
- Spectacle (id, idArtiste, idSalle, dateSpectacle, nbBilletsVendus)
- Billet (id, idSpectateur, idSpectacle, catégorie, prix)

Notez que les artistes et les spectateurs sont des personnes. Tous les attributs **id** sont des clés primaires, ceux commençant par **id** sont des clés étrangères.

11.1.1 Conception (5 points)

— Donnez le modèle entité / association correspondant au schéma de notre base.
— Pensez-vous qu'une association a été réifiée ? Si oui, laquelle ?
— Donnez la commande de création de la table `Spectacle`
— On veut qu'un même spectateur puisse avoir au plus un billet pour un spectacle. Quelle dépendance fonctionnelle cela ajoute-t-il ? Que proposez-vous pour en tenir compte dans le schéma ?

11.1.2 SQL (6 points)

Question de cours. Deux requêtes syntaxiquement différentes sont équivalentes si
— Elles peuvent donner le même résultat
— Elles donnent toujours le même résultat, quelle que soit la base
Indiquez la bonne réponse.

Donnez les expressions SQL des requêtes suivantes
— Noms des artistes âgés d'au moins 70 ans qui ont donné au moins un spectacle dans la salle "Bercy"
— Donnez le nom des artistes, des salles où ils se sont produits, triés sur le nombre de billets vendus
— Quelles personnes n'ont jamais acheté de billet ?
— Donnez la liste des spectateurs ayant acheté au moins 10 billets, affichez le nom, le nombre de billets et le prix total.

11.1.3 Algèbre (5 points)

Voici 3 relations.

A	B	C	D
1	0	1	2
4	1	2	2
6	2	6	3
7	1	1	3
1	0	1	1
1	1	1	1

Relation R

A	B	E
1	0	2
4	1	2
1	0	7
8	6	6

Relation S

A	B	E
1	0	2
4	2	2
1	0	7
8	6	5
8	5	6

Relation T

— Pour quelle requête le résultat contient-il plus d'un nuplet ? Attention : souvenez-vous que l'opérateur de projection élimine les doublons.

1. $\pi_{A,C}(\sigma_{B=0}(R))$
2. $\pi_{A,C}(\sigma_{D=0}(R))$
3. $\pi_{A,C}(\sigma_{B=0}(R) \cup \sigma_{D=0}(R))$
4. $\pi_{A,C}(\sigma_{A=C}(R)$

— Combien de nuplets retourne la requête $\pi_{A,B,E}(S \bowtie_{A=A \wedge B=B} R)$?

1. 2
2. 3
3. 4
4. 5

— Combien de nuplets retourne la requête $R \bowtie_{A=A \wedge B=B} (S \cup T)$?

1. 3
2. 5
3. 7
4. 9

— Donnez l'expression algébrique pour la requête « Noms et prénoms des spectateurs qui ont acheté au moins une fois un billet de plus de 500 euros. »

— Donnez l'expression algébrique pour la requête « Id des personnes qui ne sont pas des artistes »

11.1.4 Transactions (4 points)

On prend le schéma donné au début de l'examen.

— **Question de cours**. Quel est le principal inconvénient du mode `serializable` ?

1. Le système ne peut exécuter qu'une transaction à la fois
2. Certaines transactions sont mises en attente avant de pouvoir finir
3. Certaines transactions sont rejetées par le système

Indiquez la bonne réponse

— Exprimez au moins un critère de cohérence de cette base

— On écrit une procédure de réservation d'un billet pour un spectacle.

```
Reserver (v_idSpectateur, v_idSpectacle, v_categorie,v_ prix)
  # Requête SQL de mise à jour du spectacle
  requêteA
  # Requête SQL d'insertion du billet
 requêteB
```

Donnez les deux requêtes SQL, indiquez l'emplacement du `commit` et justifiez-les brièvement.

11.1.5 Correction de l'examen

Conception

Le schéma E/A est donné dans la Fig. 11.1. Notez que l'on ne représente pas les clés étrangères comme attributs des entités : les clés étrangères sont le mécanisme *du modèle relationnel* pour représenter les liens entre entités. Dans le *modele EA*, ces liens sont des associations, et il serait donc redondant de faire figure également les clés étrangères

Le nommage des assocations et des attributs est libre, l'important est de privilégier la clarté et la précision, notamment pour les associations.

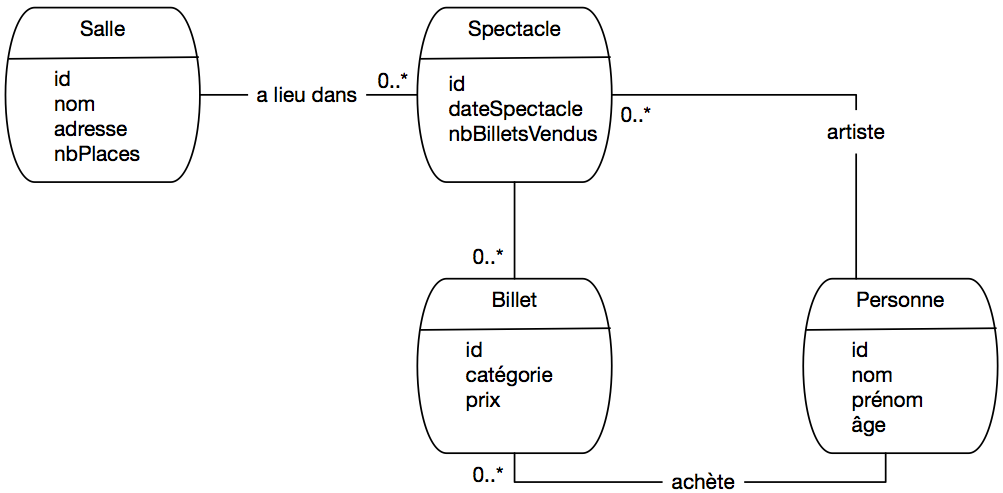

Fig. 11.1 – Le schéma E/A après rétro-conception

Le type d'entité `Billet` est issu d'une réification d'une association plusieurs-plusieurs. La principale différence est que les billets ont maintenant leur identifiant propre, alors que l'association plusieurs-plusieurs est identifiée par la paire (idSpectacle, idPersonne).

Voici la commande de création de la salle. On a choisi de mettre `not null` partout pour maximiser la qualité de la base, ce sont des contraintes que l'on peut facilement lever si nécessaire.

```
create table Spectacle (id integer not null,
       idArtiste integer not null,
       idSalle integer not null,
       dateSpectacle date not null,
```

(suite sur la page suivante)

(suite de la page précédente)

```
       nbBilletsVendus integer not null,
       primary key (id),
       foreign key (idArtiste) references Personne (id),
       foreign key (idSalle) references Salle (id)
       )
```

Dans le schéma relationnel proposé, il est possible d'avoir plusieurs billets (avec des identifiants distincts) pour le même spectacle et le même spectateur. C'est un effet de la réification : on a perdu la contrainte d'unicité sur la paire (idSpectacle, idPersonne).

On peut se poser la question d'ajouter ou non cette contrainte. Si oui, alors (idSpectacle, idSpectateur) devient une clé secondaire, que l'on déclare avec la clause `unique` dans la commande de création de la table `Billet`.

```
create table Billet (id integer not null,
       idSpectacle integer not null,
       idSpectateur integer not null,
       catégorie varchar(4) not null,
       prix float non null,
       primary key (id),
       unique (idSpectacle, idSpectateur),
       foreign key (idSpectacle) references Spectacle (id),
       foreign key (idSpectateur) references Personne (id)
       )
```

SQL

Deux requêtes sont équivalentes si elles donnent toujours le même résultat, quelle que soit la base.

```
select p.nom
from Personne as p, Spectacle s, Salle l
where p.id = s.idArtiste and s.idSalle = l.id
and l.nom = 'Bercy' and p.age >= 70
```

```
select p.nom as nomArtiste, s.nom as nomSalle, sp.nbBilletsVendus
from Peronne as p, Spectacle as sp, Salle as s
where p.id = sp.idArtiste
and sp.idSalle = s.id
order by sp.nbBilletsVendus
```

```
select prénom, nom
from Personne as p
where not exists (select '' from Billet where idSpectateur = p.id)
```

```
select p.nom, count(*), sum(prix)
from Personne as p, Billet as b
```

(suite sur la page suivante)

(suite de la page précédente)

```
where p.id = b.idSpectateur
group by p.id, p.nom
having count(*) >= 10
```

Algèbre

— La requête 4 renvoie 2 nuplets : (1,1) et (6,6)
— Voici l'union $S \cup T$:

A	B	E
1	0	2
4	1	2
8	6	6
4	2	2
1	0	7
8	6	5
8	5	6

La jointure sur A et B avec R renvoie 5 nuplets : (1,0,1,2) s'associe à 2 nuplets dans $S \cup T$, de même que (1,0,1,1). (4,1,2,2) s'associe à un nuplet.

Relation T

— La requête renvoie également 3 nuplets
— $\pi_{nom,prenom}(Personne \underset{id=idSpectateur}{\bowtie} \sigma_{prix>500}(BILLET))$
— $\pi_{id}(Personne) - \pi_{idArtiste}(Spectacle)$

Transactions

Le mode sérialisable entraîne des rejets de transactions.

Le nombre de billets vendus (`nbBilletsVendus` dans `Spectacle`) pour un spectacle est égal au nombre de lignes dans `Billet` pour ce même spectacle.

Autre possibilité : pas plus de billets vendus que de places dans la salle.

Un `update` de `Spectacle` pour incrémenter `nbBilletsVendus`,

```
update Spectacle set nbBilletsVendus=nbBilletsVendus+1 where idSpectacle=v_
↪idSpectacle
```

et un `insert` dans `Billet`.

```
insert into Billet (id, idSpectateur, idSpectacle, catégorie, prix)
value (1234, v_idSpectateur, v_idSpectacle, v_catégorie, v_prix)
```

Le `commit` vient à la fin car c'est seulement à ce moment que la base est à nouveau dans un état cohérent.

11.2 Examen session 2, avril 2019

Soit la base de données suivante permettant de gérer un championnat de football.

- Stade(**id**, ville, nom, nbPlaces, prixBillet)
- Equipe(**id**, pays, siteWeb, entraîneur)
- Joueur(**id**, *idEquipe*, nom, prénom, âge)
- Match(**id**, *idStade*, dateMatch, *idEquipe1*, *idEquipe2*, scoreEquipe1, scoreEquipe2, nbBilletsVendus)
- But(**idJoueur, idMatch, minute**, penalty)

Tous les attributs **en gras** sont des clés primaires, ceux *en italiques* sont des clés étrangères. Notez qu'une clé étrangère peut faire partie d'une clé primaire.

Chaque pays a une seule équipe en compétition. Une ville peut avoir plusieurs stades. L'attribut `But.penalty` vaut VRAI si le but a été marqué suite a un penalty (coup de pied de réparation) ou FAUX dans le cas contraire. Le nom de l'entraîneur d'une équipe est donné par l'attribut `Equipe(entraîneur)`.

11.2.1 Conception (6 points)

Etudions la conception de cette base en répondant aux questions suivantes.

- Avec ce schéma, une équipe peut-elle jouer contre elle-même ? Expliquez.
- Avec ce schéma, un joueur peut-il marquer un but dans un match auquel son équipe ne participe pas ? Expliquez.
- Est-il possible de savoir dans quelle ville a été marqué chaque but ? Expliquez.
- Donnez le schéma entité-association correspondant aux relations `Match`, `Equipe`, `Stade`.
- Donnez la commande SQL de création de la table `Match`.
- Citer au moins une clé *candidate* autre que la clé primaire parmi les attributs des tables (bien lire l'énoncé). Quel serait l'impact si on la choisissait comme clé primaire ?
- Un de vos collègues propose de modéliser un but comme une association plusieurs-plusieurs entre un joueur et un match. Cela vous semble-t-il une bonne idée ? Cela correspond-il au schéma relationnel de notre base ?

11.2.2 SQL (8 points)

Exprimez les requêtes suivantes en SQL.

- Noms des joueurs âgés de plus de 30 ans qui ont marqué un but dans la première minute de jeu.
- Noms des joueurs français qui n'ont marqué aucun but
- Le prix du billet le plus cher
- Donnez le nom des entraineurs avec le nombre de buts marqués par leur équipe
- Donnez le nom des joueurs ayant marqué un but dans un match auquel leur équipe ne participe pas.
- Les noms des joueurs français qui ont marqué un but lors d'un match entre la France et la Suisse joué à Lille. Attention : dans la table `Match` la France peut être l'équipe 1 ou l'équipe 2. Il faut construire la requête qui correspond aux deux cas.

11.2.3 Algèbre (3 points)

— Donnez l'expression algébrique de la première requête SQL (section précédente)
— Donnez l'expression algébrique pour la seconde requête (les joueurs français qui ne marquent pas)
— Donnez la requête SQL correspondant à l'expression algébrique suivante et expliquez le sens de cette requête

$$\pi_{idMatch}(But \bowtie_{idMatch=id} \sigma_{scoreEquipe1=0 \wedge scoreEquipe2=0}(Match))$$

11.2.4 Programmation et transactions (3 points)

— Parmi les phrases suivantes, lesquelles vous semblent exprimer une contrainte de *cohérence* correcte sur la base de données ?
  — Pour chaque match nul, je dois trouver autant de lignes associées au match dans la table `But` pour l'équipe 1 et pour l'équipe 2.
  — Quand j'insère une ligne dans la table `But` pour l'équipe 1, je dois effectuer également une insertion pour l'équipe 2 dans la table `But`.
  — Si je trouve une ligne dans la table `Match` avec `id=x`, `scoreEquipe1=y` et `scoreEquipe2=z`, alors je dois trouver `y+z` lignes dans la table `But` avec `idMatch=x`.
— Voici un programme en pseudo-code simplifié à exécuter chaque fois qu'un but est marqué dans un match, par un joueur et au bénéfice de la première équipe. Les "#" marquent des commentaires.

```
function marquer_but_eq1 (imatch, ijoueur, min, peno)
   startTransaction
   # A
   select scoreEquipe1 into :scoreE where idMatch = :imatch
   #B
   update Match set scoreEquipe1 = :scoreE + 1 where idMatch =␣
↪:imatch
   # C
   insert into But (idMatch, idJoueur, minute, penalty)
   values (:idMatch, :ijoueur, :min, :peno)
   # D
```

Où faut-il selon-vous ajouter un ordre `commit` ?
— Juste après `B` et `C`
— Juste après `A`
— Juste après `D`
Justifiez votre réponse.

Supposons que je lance deux exécutions de `marquer_but_eq1` en même temps. Dans quel scénario l'exécution imbriquée amène-t-elle une incohérence ?

11.2.5 Corrigé

Conception

— Oui, la base permet de représenter un but marqué dans un match opposant deux équipes dont aucune n'est celle du joueur.
— Oui, par transitivité $But \rightarrow Match$, $Match \rightarrow Stade$ et $Stade \rightarrow Ville$.
— Schéma

```
CREATE TABLE Match(
       id NUMBER(10),
      idStade NUMBER(10),
      dateMatch DATE,
      idEquipe1 NUMBER(10),
      idEquipe2 NUMBER(10),
      scoreEquipe1 NUMBER(10),
      scoreEquipe2 NUMBER(10),
      PRIMARY key (id),
      foreign key (idStade) REFERENCES Stade(id),
      foreign key (idEquipe1) REFERENCES Equipe(id),
      foreign key (idEquipe2) REFERENCES Equipe(id)
);
```

— Le pays est clé candidate pour l'équipe. Il faudrait alors l'utiliser comme clé étrangère partout.

SQL

```
select nom
from Joueur as j, But as b
where j.id = b.idJoueur and j.age > 30 and b.minute = 1;

select nom
from Joueur as j, Equipe as e
where j.idEquipe = e.id
and e.pays = 'France'
and j.id not IN (select idJoueur from But);

select prixBillet
from Stade as s1
where not exists (select ''
           from Stade as s2
           where s2.prixBillet > s1.prixBillet);

select entraineur, count(*)
from Equipe as e, Joueur as j, But as b
where e.id=j.idEquipe
and b.idJoueur = j.id
```

(suite sur la page suivante)

(suite de la page précédente)

```
group by e.id, e.entraineur;

select prénom, nom
from Match as m, Joueur as j, But as b
where e.id=j.idEquipe
and b.idJoueur = j.id
and m.id = b.idMatch
and e.id != m.idEquipe1
and e.id != e.idEquipe2
```

Il faut soit Match.equipe1=”France”, soit Match.equipe2=”France”.

```
select j.nom
from Match m, Equipe e1, Equipe e2, Joueur j, Stade s, But b
where m.idEquipe1 = e1.id
and m.idEquipe2 = e2
and ((e1.pays = 'France' id and e2.pays='Suisse' and e1.id = j.idEquipe)
      or
  (e2.pays = 'France' id and e1.pays='Suisse' and e2.id = j.idEquipe))
and m.idStade = s.id
and s.ville = 'Lille'
and b.idJoueur = j.id
and b.idMatch = m.id )
```

Algèbre

$$\pi_{nom}(\sigma_{age>30}(Joueur) \underset{id=idJoueur}{\bowtie} \sigma_{minute=1}(But))$$

Joueurs français :

$$A = \sigma_{pays='France'}(Joueur\ j) \underset{j.idEquipe=e.id}{\bowtie} Equipe\ e)$$

Joueurs français qui ont marqué au moins un but :

$$B = \sigma_{pays='France'}(But\ b \underset{b.idJoueur=j.id}{\bowtie} Joueur\ j) \underset{j.idEquipe=e.id}{\bowtie} Equipe\ e)$$

Résultat :

$$Resultat := A - B$$

Les matchs nuls 0-0 pour lequel il existe quand même un but.

```
select idMatch
from But as b, Match as m
where b.idMatch=m.id
and m.scoreEquipe1=0
and m.scoreEquipe2=0
```

11.3 Examen session 1, présentiel, juillet 2019

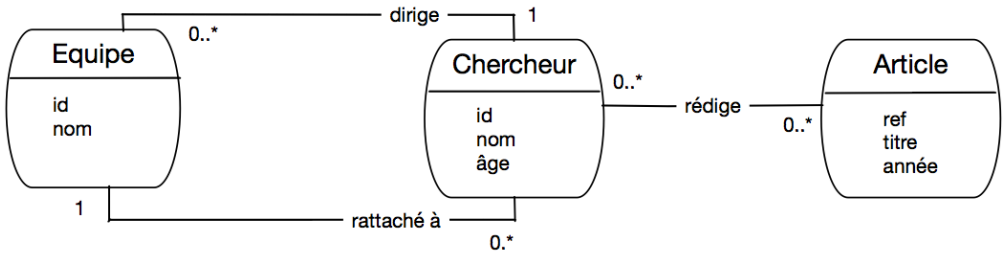

Fig. 11.2 – Un laboratoire d'informatique

On cherche à modéliser un laboratoire d'informatique afin de pouvoir gérer ses équipes, ses chercheurs et leurs publications (articles). L'analyse donne le schéma entité/association de la Fig. 11.2. À partir de cette analyse, quelqu'un propose le schéma relationnel suivant, dans lequel les clés primaires sont en gras (à vous de trouver les clés étrangères) :

— Equipe (**id**, nom, idDirecteur)
— Chercheur (**id**, nom, âge, idEquipe)
— Article (**réf**, titre, année)
— Rédige (**idChercheur**, réfArticle)

11.3.1 Conception (6 points)

— Ce schéma relationnel représente-t-il correctement le modèle conceptuel de la Fig. 11.2 ? Vérifiez les clés primaires et étrangères et proposez éventuellement des corrections.
— On veut ajouter la position d'un chercheur dans l'ordre des auteurs pour un article. Où placer cette information, dans le modèle entité association et dans le modèle relationnel ?
— Un article peut-il être rédigé par des chercheurs appartenant à des équipes différentes (justifiez) ?
— Un chercheur peut-il diriger une équipe à laquelle il n'est pas rattaché (justifiez) ?
— Donnez les commandes `create table` pour les tables `Chercheur`, `Article` et `Rédige` (en tenant compte des corrections éventuelles de la question 1).

11.3.2 SQL (8 points)

Exprimez les requêtes suivantes en SQL.

— Noms des chercheurs de l'équipe nommée `Vertigo`
— Titre des articles parus depuis 2015 dont l'un au moins des auteurs appartient à l'équipe `ROC`
— Quels articles n'ont pas d'auteur ?
— Titre des articles parus depuis 2015 dont tous les auteurs appartiennent à l'équipe `ROC` (aide : la quantification universelle se remplace par la quantification existentielle et la négation)
— Nom des chercheurs qui dirigent une équipe à laquelle ils n'appartiennent pas
— Nom des chercheurs qui dirigent plus d'une équipe.

11.3.3 Algèbre (3 points)

— Donnez l'expression algébrique pour les deux premières requêtes SQL (section précédente)
— Donnez la requête SQL correspondant à l'expression algébrique suivante et expliquez-en le sens.

$$\pi_{id}(Chercheur) - \pi_{idChercheur}(Rédige)$$

11.3.4 Valeurs nulles (2 pts)

Le tableau suivant montre une instance de `Article`. Les cellules blanches indiquent des valeurs inconnues.

réf	titre	année
AR243	Les ordinateurs pensent-ils ?	2018
AR254	Money for nothing	
AR20		2010

Donnez le résultat des requêtes suivantes :

```
select réf from Article where année > 2015;

select réf from Article where année <= 2015;

select réf from Article where annee > 0 or titre is not null;

select réf from Article where (année < 2015 or année IS null)
    and titre like `%`
```

11.3.5 Transactions (1 pt)

Soit deux transactions concurrentes T_1 et T_2. Quelle(s) affirmation(s), parmi les suivantes, est (sont) vraie(s) dans un système transactionnel assurant les propriétés ACID en isolation complète.

— Si T_2 débute après T_1, T_1 ne voit jamais les mises à jour de T_2.
— T_1 ne voit pas ses propres mises à jour tant qu'elle n'a pas validé.
— Si T_2 débute après T_1, T_1 ne voit les mises à jour de T_2 qu'après que T_2 a effectué un `commit`.
— Si T_1 et T_2 veulent modifier le même nuplet, cela déclenche un interblocage (`deadlock`).

11.4 Examen session 2, présentiel, septembre 2020

Voici une partie de la base de données utilisée pour gérer des œuvres dans un musée :

— Oeuvre(**idOeuvre**, nom, idPropriétaire, idAuteur)
— Personne(**idPersonne**, nom, prénom)
— Expertise(**idExpertise**, idOeuvre, idExpert, valeur)

Les auteurs, les propriétaires et les experts sont des personnes bien entendu. Les clés primaires sont en gras, les clés primaires ne sont pas explicitement indiquées.

11.4.1 Schéma relationnel (6 points)

— Pour chaque table, donner les dépendances fonctionnelles liant les clés primaires et clés étrangères (1 point)
— En déduire les associations de type *plusieurs à 1* du schéma Entité/Association correspondant à ce schéma relationnel (1 point)
— Donner ce schéma Entité/Association complet (1 point)
— Donner des commandes SQL permettant de créer les tables `Oeuvre` et `Personne` (1 point)
— J'ajoute la contrainte *Un expert ne peut évaluer une œuvre qu'une seule fois*. Quelle est la dépendance fonctionnelle correspondante et quel changement du schéma relationnel proposez-vous ? (1 point)
— Ce schéma permet-il de représenter le fait qu'une personne peut être à la fois propriétaire et auteur d'une même œuvre ? Argumentez votre réponse. (1 point)

11.4.2 SQL (7 points)

— Donner le nom du propriétaire et le nom de l'auteur pour l'œuvre dont l'identifiant est "fg65".
— Donner les noms des œuvres expertisées par leur propriétaire
— Donner les noms et prénoms des propriétaires d'une œuvre qui n'ont jamais effectué d'expertise.
— Donner les œuvres et le nom des personnes qui n'en sont ni propriétaire, ni auteur.
— Donner les noms et prénoms des personnes qui sont à la fois auteur, propriétaire et expert (mais pas forcément de la même œuvre).
— Pour chaque œuvre donnez la moyenne des valeurs estimées par les experts.
— Donnez le nombre d'expertises pour les œuvres dont la valeur maximale estimée est de 10 000 Euros

11.4.3 Un peu de logique (3 points)

On dispose de deux prédicats $Auteur(x)$ et $Prop(x)$ qui sont vrais si, respectivement, x est auteur ou x est propriétaire.
— Quelle formule exprime la condition « Soit x est propriétaire, soit x est auteur mais pas les deux »
— Quelle est la négation de l'énoncé : « Soit x est propriétaire, soit x est auteur »
— Comment exprimer l'énoncé suivant sans implication : « Si x est auteur, alors x n'est pas propriétaire » (utiliser uniquement les connecteurs de SQL : and, or et not).

11.4.4 Algèbre (2 points)

On dispose de deux tables $T_1(A, B, C)$ et $T_2(D, E, F)$. Donnez une expression algébrique équivalente à la suivante, avec les contraintes suivantes : on peut utiliser la jointure mais pas le produit cartésien, et une sélection doit s'appppliquer directement à une table.

$$\sigma_{A=C \wedge A=D \wedge E=F \wedge C>B}(T_1 \times T_2)$$

11.4.5 Transactions (2 points)

On dispose d'une table $T(id, valeur)$. Initialement toutes les valeurs sont différentes. Voici une procédure qui échange les valeurs de 2 nuplets.

```
create or replace procedure Echange (id1 INT, id2 INT) AS
   -- Déclaration des variables
       val1, val2 integer;
   begin
       -- On recherche la valeur de id1 et de id2
       select valeur into val1 from T where id = id1
       select valeur into val2 from T where id = id2

       -- On échange les valeurs
       update T SET valeur = val1 where id = id2
       update T SET valeur = val2 where id = id1

    end;
```

On est en mode `Autocommit` : un `commit` a lieu après chaque requête SQL. Expliquez dans quel scénario l'exécution concurrente de deux procédures d'échange peut aboutir à ce que deux nuplets aient la même valeur.

11.5 Examen session 1, FOAD, janvier 2022

Le schéma de base de données suivant sera utilisé pour l'ensemble du sujet. Il permet de gérer les souscriptions pour un ensemble d'opérateurs de téléphonie mobile. Ce schéma sera nommé **schéma final** par la suite.

- Opérateur (**id**, nom)
- Forfait (**id**, nom, idOpérateur, prix)
- Client(**id**, nom, prénom, ville)
- Souscription(**idClient, idForfait**, durée, numéro)

L'attribut *durée* est un entier positif exprimant le nombre de mois d'engagement. les clés étrangères ne sont pas indiquées.

11.5.1 Conception (8 points)

La Fig. 11.3 montre une modélisation initiale de la base par un schéma entité association.

Questions :

- Donnez la liste des dépendances fonctionnelles définies par ce schéma initial (1 pt)
- Donnez le schéma relationnel correspondant à la Fig. 11.3, sous forme simplifiée (Nom de table, liste des attributs en encadrant les clés primaires et en soulignant les clés étrangères) (1 pt)
- Quelles sont les différences entre le schéma initial et le schéma final ? À quelle(s) évolution(s) des choix de modélisation ce changement correspond-t-il (2 pts) ?

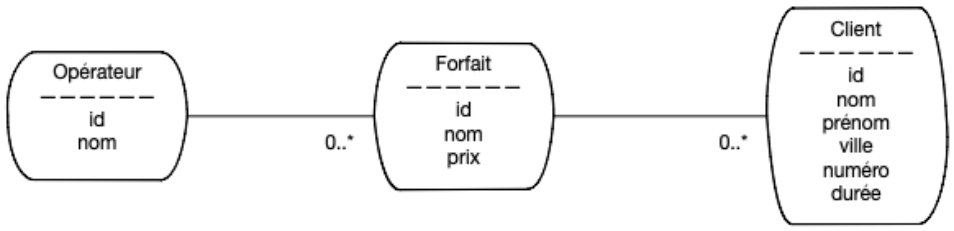

Fig. 11.3 – Modélisation initiale de la base

— Dessinez le schéma final sous forme entité-association. Quelles dépendances fonctionnelles ont changé par rapport au schéma initial ? (1 pt)
— Dans le schéma final, une réification serait-elle possible ? Quels changements impliquerait-elle ? (1 pt) ?
— Donnez les commandes `create table` pour le schéma final (2 pts)

11.5.2 SQL (7 points)

Exprimez en SQL les requêtes suivantes **sur le schéma final, donné dans l'énoncé de l'examen.**

— Nom et prénom des clients qui ont souscrit un forfait avec engagement de plus de 24 mois.
— Existe-t-il deux clients qui auraient le même numéro de téléphone pour le même forfait ? Donnez leurs noms et prénoms (des .clients)

— Noms des clients qui ont un forfait nommé `Audace` et un autre nommé `Privilège`.

— Quels clients n'ont pas souscrit de forfait ?
— Quels opérateurs n'ont pas de client à Lyon ?
— Donnez le nombre de souscriptions pour chaque forfait de l'opérateur `Violet`.
— Quels clients ont deux souscriptions ou plus ? Donnez le nombre de souscriptions.

11.5.3 Algèbre relationnelle (3 pts)

— Expliquez ce que fait la requête algébrique suivante, et donnez une expression SQL équivalente

$$\pi_{nom,prenom}(\sigma_{ville='Paris'}(Client) \underset{id=idClient}{\bowtie} (\sigma_{duree>24}(Souscription) \underset{idForfait=id}{\bowtie} \sigma_{nom='\text{Audace}'}(Forfait))$$

— Même question pour l'expression suivante

$$\pi_{c.id,c.nom,c.prenom}(Client) - \pi_{c.id,c.nom,c.prenom}(Client \underset{c.id=s.idClient}{\bowtie} \sigma_{duree\geq 48}(Souscription))$$

— Voici une expression SQL « algébrique »

```
select c.nom, c.prénom
from (Client as c join Souscription as s on c.id=s.idClient)
join (select * from Forfait as f where nom= 'Audace')
 on s.idForfait = f.id
```

Donnez une expression SQL « déclarative » équivalente, et sans requête imbriquée.

11.5.4 Jointure externe (2 pts)

Que renvoie la requête suivante ?

```
select c.nom, c.prénom, f.nom
from (Client as c left outer join (Souscription as s join Forfait as f
                                   on s.idForfait=f.id)
    on c.id = s.idClient
where ville= 'Nantes'
```

11.6 Examen session 1, second semestre 2023

Une application met en contact des personnes dont certaines (les professeurs) proposent des leçons en ligne à d'autres (les étudiants). Dans ce qui suit on va supposer ques les leçons sont des cours de langue.

Au départ le concepteur a placé les données sur les leçons dans un unique fichier CSV dont voici un extrait du contenu sous forme de tableau.

idProf	professeur	idEtudiant	étudiant	langue	date
1	Luca	2	Philippe	italien	14 mars
2	Philippe	3	Lydia	français	15 mars
1	Luca	3	Lydia	italien	20 mars
2	Philippe	3	Lydia	français	2 avril
1	Luca	3	Lydia	italien	6 avril
3	Lydia	2	Philippe	anglais	12 avril
3	Lydia	2	Philippe	allemand	13 avril

On va chercher à faire mieux avec une base relationnelle.

11.6.1 Conception du schéma (7 pts)

— Pour chacune des DFs suivantes, indiquez si elle est respectée ou pas *dans le tableau précédent*. Si elle ne l'est pas, expliquez pourquoi.

  — $professeur \rightarrow langue$
  — $professeur, date \rightarrow etudiant$
  — $etudiant \rightarrow professeur$
  — $etudiant, langue \rightarrow professeur$

— Finalement, après avoir étudié beaucoup plus d'exemples, on détermine les dépendances fonctionnelles suivantes : $idProf \rightarrow professeur$, $idEtudiant \rightarrow etudiant$ et $(idProf, idEtudiant, date) \rightarrow langue$.
  — Quelle est la clé de la relation ?
  — Quelle est la décomposition en 3FN ?

Pour la suite de l'examen nous travaillons avec le schéma suivant.

— Personne(**id**, nom, codeLangueNatale)
— Langue (**code**, intitulé)
— Cours(**id**, idProfesseur, idEtudiant, codeLangue, date)
— Sur ce schéma, répondez aux questions suivantes :
  — Enumérez les clés étrangères
  — Donnez le schéma entité-association correspondant
  — Donnez les commandes `create table`
  — Une des relations correspond-elle à une réification ? Si oui indiquez ce que serait le schéma sans réification.

11.6.2 SQL (7 pts)

Sur le schéma donné précédemment, exprimez en SQL les requêtes suivantes :

— Liste des cours de l'étudiant Philippe (on suppose qu'il n'y en a qu'un), en indiquant le nom du professeur et la date.

  — Les personnes qui sont professeur *et* étudiant

  — Les professeurs qui donnent des cours dans une autre langue que leur langue natale

  — **Les professeurs qui ne donnent jamais cours dans une autre langue** que leur langue natale

  — Les personnes et les langues qu'elles n'apprennent pas

  — Donner, pour chaque personne, le nombre de langues qu'elle apprend

  — **Donner les langues pour lesquelles on trouve au moins** trois personnes dont c'est la langue natale.

11.6.3 Algèbre (2 pts)

Donnez les expressions algébriques pour les requêtes suivantes :

— Nom des personnes qui se donnent un cours à elles-mêmes..
— Les personnes (on se contentera de l'identifiant) qui n'étudient pas l'italien (langue dont le code est `it`)

11.6.4 Valeurs nulles, vues (2 pts)

— Créez une vue `Inscriptions` donnant, pour chaque personne, le code des langues apprises, ou `NULL` si aucune langue n'est apprise.
— En utilisant cette vue, quelle requête donne les personnes qui n'apprennent aucune langue ?

11.6.5 Procédures (2 pts)

Expliquez ce que signifie l'immuabilité d'un curseur.

CHAPITRE 12

Indices and tables

— genindex
— modindex
— search